Navigierbare Kompression von XML-Datenströmen

Dissertation zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Fakultät für Elektrotechnik, Informatik und
Mathematik der Universität Paderborn

vorgelegt von
Rita Hartel
Paderborn, September 2008

Abstract

Nowadays, XML has shown to be the de facto standard for electronic data interchange on the Internet. Available XML data ranges from small Web pages to possibly unbounded streams, as used e.g. in news agencies.

Especially when using small mobile devices (such as mobile phones or PDAs), the data size forms a problem due to the limitations in main memory, and the size of the transferred data forms a problem due to the limitations in energy consumption. In these cases, i.e., whenever the data size or energy consumption limitations form the bottleneck of an XML based application, these applications can profit from the usage of XML compression. It is desirable that these applications can perform all XML based operations, like XML query evaluation and XML data manipulation, directly on the compressed XML data, to avoid additional computation caused by prior decompression and subsequent compression. Furthermore, it is desirable that there is no loss in efficiency when performing query evaluation on compressed data in contrast to performing query evalution on uncompressed data. Finally, it is desirable that it is possible to perform the query evaluation on possibly unbounded streams.

The existing approaches to XML compression can be classified according to whether they support these XML based operations or not. Approaches like GZip, BZip2, XMill, and others reach a strong compression, while the evaluation of queries requires prior decompression and subsequent compression. Other approaches, like XGrind and XQueC, allow to evaluate queries on the compressed data, but they are outperformed in terms of compression ratio by those approaches that do not support these XML based operations. The approaches to XML compression that where implemented and presented in this thesis allow query evaluation and updates on the compressed data while at the same time, they reach compression ratios comparable to those approaches that do not support the XML based operations. The approaches presented in this thesis focus on XML structure compression, i.e., on the compression of the XML elements and attributes, but all presented approaches can be combined arbitrarily with quite a number of data compressors for text and attribute values that are based on prior ideas from literature.

In this thesis, I present three basic approaches to the compression of XML structure – coding based compression, compression based on structural redundancies, and schema based compression – and demonstrate that query evaluation as well as update operations can be performed directly on the resulting compressed data. The presented approaches provide compression ratios significantly better than those of other queriable XML compression techniques like e.g. XGrind. Futhermore, I present two combinations of these compression approaches that provide the advantages of the combined approaches and thereby reach a better compression ratio and a faster compression and decompression while showing the same capabilities in other respects.

The approaches to XML structure compression presented in this thesis support query evaluation in form of a simple interface. To support all axes of XPath, these approaches are amended by a generic approach to XPath query evaluation. This approach allows to efficiently evaluate queries on compressed XML representations that implement a simple, shallow interface. Amongst compressed XML, this approach allows to evaluate queries on uncompressed XML files and on possibly unbounded data streams. When comparing this approach with the standard XPath evaluator JAXP, the generic XPath evaluator reached evaluation times comparable or even better than those of JAXP.

Finally, I have performed extensive performance evaluations to compare the presented compression approaches with other available approaches to XML compression, and I show that the presented compression approaches outperform the other approaches GZip, BZip2, and XMill in terms of query evaluation und updates on the compressed data.

When comparing the newly developed approaches to XML compression with each other, it can be seen that each approach shows its strength in different aspects: While one approach reaches a strong compression, another approach shows fast compression and decompression times, whereas the third approach allows to evaluate queries efficiently. None of the presented approaches outperforms the other approaches completely.

Zusammenfassung

Heutzutage hat sich XML als der de facto Standard für den Datenaustausch im Internet durchgesetzt. Dabei reicht die Spanne an verfügbaren XML-Daten von kleinen Webseiten bis hin zu möglicherweise unendlichen Datenströmen, wie sie z.B. von Nachrichten-Agenturen versandt werden.

Vor allem beim Einsatz von mobilen Kleinstgeräten (wie z.B. Mobiltelefonen oder PDAs) stellt die Datengröße aufgrund des nur eingeschränkt verfügbaren Arbeitsspeichers und die Übertragungsgröße aufgrund der nur eingeschränkt verfügbaren Energie ein Problem dar. In diesen Fällen, also immer, wenn Datengröße oder begrenzter Energieverbrauch das Nadelöhr einer XML-basierten Anwendung darstellen, können diese Anwendungen von der Nutzung von XML-Kompression profitieren. Hierbei ist es wünschenswert, dass diese Anwendungen alle XML-basierten Operationen, wie z.B. XML-Anfrage-Auswertung oder Manipulationen der XML-Daten, direkt auf den komprimierten Daten durchführen können, um einen Mehraufwand durch vorherige Dekompression und anschließende Kompression zu vermeiden. Weiterhin ist es wünschenswert, dass bei der Anfrage-Auswertung auf komprimierten Daten keine Effizienzverluste im Vergleich zur Anfrage-Auswertung auf nicht-komprimierten Daten auftreten, und dass diese Anfrage-Auswertung auch auf quasi-unendlichen komprimierten Datenströmen möglich ist.

Bisher in der Literatur verfügbare Kompressions-Verfahren lassen sich unter anderem danach gliedern, ob sie die XML-basierten Operationen unterstützen oder nicht. Verfahren wie GZip, BZip2, XMill und andere erreichen eine sehr starke Kompression, die Durchführung der XML-basierten Operationen erfordert jedoch vorherige Dekompression und anschließende Kompression. Andere Verfahren wie z.B. XGrind oder XQueC bieten zwar die Möglichkeit, Anfragen direkt auf dem Komprimat auszuwerten, sie bleiben jedoch bezüglich ihrer Kompressionsstärke deutlich hinter denjenigen Kompressoren zurück, die die XML-basierten Operationen nicht unterstützen. Die in dieser Arbeit entwickelten und vorgestellten Kompressions-Verfahren bieten im Gegensatz dazu die Möglichkeit, Anfrage-Auswertung und Updates direkt auf dem Komprimat durchzuführen, wobei gleichzeitig Kompressionsstärken vergleichbar mit denen derjenigen Kompressoren erreicht werden, die diese XML-basierten Operatio-

nen nicht unterstützen. Hierbei konzentriert sich diese Arbeit insbesondere auf die Struktur-Kompression, also auf die Kompression der XML-Elemente und -Attribute, wobei alle vorgestellten Struktur-Kompressions-Verfahren beliebig mit einer Reihe von Daten-Kompressoren kombiniert werden können, welche auf bereits in der Literatur diskutierten Ideen basieren.

In dieser Arbeit werden drei grundlegende Verfahren zur XML-Struktur-Kompression – Kodierungs-basierte Kompression, Kompression basierend auf Struktur-Redundanzen sowie Schema-basierte Kompression – präsentiert, und es wird nachgewiesen, dass sowohl Anfrage-Auswertung als auch Updates direkt, also ohne Dekompression, auf den aus diesen Verfahren resultierenden Komprimaten durchgeführt werden können. Diese Verfahren erreichen bessere Kompressionsraten als Kompressions-Verfahren mit vergleichbaren Eigenschaften bei der Anfrage-Auswertung wie z.B. XGrind. Des Weiteren werden zwei Kombinationsmöglichkeiten dieser Verfahren vorgestellt, die die Vorteile der jeweiligen Verfahren vereinigen, und somit bei ansonsten gleichbleibenden Eigenschaften gleichzeitig eine stärkere Kompression und eine schnellere Dekompression erreichen.

Die in dieser Arbeit vorgestellten Verfahren zur XML-Struktur-Kompression unterstützen eine Basis-Anfrage-Auswertung in Form einer einfachen Schnittstelle. Um alle Achsen des XPath-Standards zu unterstützen, wurden diese Verfahren ergänzt durch ein neu entwickeltes generisches XPath-Auswertungs-Verfahren. Dieses erlaubt effiziente Anfrage-Auswertung auf komprimierten XML-Repräsentationen, welche eine schlanke, einfache Schnittstelle implementieren. Neben komprimiertem XML, erlaubt dieses Verfahren auch die Anfrage-Auswertung auf nicht-komprimiertem XML-Dateien und auf nicht-komprimierten quasi-unendlichen Datenströmen. Bei einem Vergleich mit dem Standard-XPath-Auswerter JAXP hat sich hierbei herausgestellt, dass bei der Verwendung des neu entwickelten XPath-Auswertungs-Verfahrens Auswertungszeiten erreicht wurden, welche vergleichbar oder sogar besser waren als die von JAXP.

In einer ausführlichen Messreihe wurden die vorgestellten Verfahren untereinander und mit anderen verfügbaren XML-Kompressions-Verfahren verglichen, und es wurde nachgewiesen, dass sie den anderen Verfahren GZip, BZip2 und XMill in Bezug auf Anfrage-Auswertung und Updates überlegen sind.

Im Vergleich der neu entwickelten Kompressions-Verfahren untereinander hat sich gezeigt, dass jedes der Verfahren seine Stärke in einem anderen Bereich hat: Während eines besonders stark komprimiert, weist ein anderes besonders niedrige Kompressions- und Dekompressionszeiten vor, wohingegen das dritte Verfahren besonders effiziente Anfrage-Auswertung erlaubt. Keines der neu entwickelten und vorgestellten Verfahren ist also den jeweils anderen absolut überlegen.

Danksagung

Ohne tatkräftige Unterstützung und Hilfe Anderer hätte diese Arbeit nicht entstehen können. An dieser Stelle möchte ich mich bei all denen bedanken, die es mir ermöglicht haben, diese Dissertation anzufertigen.

Ganz besonders bedanken möchte ich mich bei Professor Stefan Böttcher, der nicht nur diese Arbeit wissenschaftlich betreut hat, sondern der mir vor allem als wertvoller Gesprächspartner bei allen fachlichen Fragen, Problemen und Diskussionen hilfreich und konstruktiv zur Seite gestanden hat.

Danken möchte ich auch Professor Gregor Engels, der nicht nur als Zweitgutachter meiner Arbeit fungiert hat, sondern mir im Vorfeld noch wertvolle Tipps zur Verbesserung dieser Arbeit gegeben hat.

Danke auch an die vielen Studenten, die durch ihre Diplom-, Studien- oder Bachelorarbeiten an der Umsetzung meiner Ideen mitgearbeitet haben.

Besonderer Dank gilt meinen Eltern, die mir diese Ausbildung erst ermöglicht haben, und die mir geholfen haben, meiner Arbeit in Bezug auf Rechtschreibung den letzten Schliff zu geben.

Insbesondere danken möchte ich aber meinem Mann Arthur, der mich während der ganzen Promotionszeit mit viel Liebe und Geduld unterstützt hat, und mich während Durststrecken immer wieder seelisch aufgebaut hat.

Rita Hartel
Paderborn, September 2008

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Im Laufe der letzten Jahre setzte sich XML immer mehr als Standard-Format zum Datenaustausch, insbesondere bei Web-Anwendungen, durch.

Während XML durch seine Anreicherung der Daten mit Struktur und der damit verbundenen Flexibilität große Vorteile gegenüber anderen Austauschformaten bietet, sorgt jedoch gerade diese Struktur für einen erheblichen Overhead im Vergleich zu den eigentlichen Nutzdaten.

Obwohl die verfügbare Bandbreite zur Datenübertragung erheblich angestiegen ist bei gleichzeitiger Reduktion der Kosten, stellt genau diese Datenübertragung, verglichen z.B. mit der Speicherung auf Festplatten oder dauerhaften Speichermedien, noch immer das Nadelöhr einer jeden Internet-basierten Anwendung dar. Insbesondere bei der Verwendung von mobilen Kleinstgeräten (wie z.B. Mobiltelefonen oder PDAs) stellen die Datenübertragung und der damit verbundene Energie-Verbrauch den größten „Kostenfaktor“ dar.

Auch der Arbeitsspeicher bei diesen mobilen Kleinstgeräten entspricht im Allgemeinen nur einem Bruchteil des Arbeitsspeichers von herkömmlichen PCs. Daher können diese Kleinstgeräte nur einen deutlich geringeren Teil eines XML-Dokumentes einsehen und bearbeiten, als dies auf einem PC möglich wäre. Bisherige Speichermodelle, wie z.B. der DOM-Baum, sind daher für solche Anwendungen auf Kleinstgeräten nur sehr eingeschränkt nutzbar.

Als eine mögliche Lösung dieser Probleme stelle ich in dieser Arbeit verschiedene Verfahren zur Kompression von XML-Dokumenten vor. Durch die verkleinerte Darstellung derselben Inhalte können sowohl der Datentransfer als auch der Arbeitsspeicher-Bedarf minimiert werden, ohne dass die durch die Semi-Strukturiertheit erzielte Flexibilität eingeschränkt werden muss (lediglich die direkte Lesbarkeit durch Menschen geht verloren). Bieten solche Verfahren zur XML-Kompression dieselben Zugriffs- und Manipulationsmöglichkeiten wie herkömmliches XML, so können alle XML-basierten Anwendungen ohne spür-

baren Nachteil auf der komprimierten Darstellung statt auf herkömmlichem XML ausgeführt werden.

1.2 Szenarien

In diesem Kapitel werde ich Szenarien vorstellen, die davon profitieren, wenn statt herkömmlichem XML eine komprimierte XML-Repräsentation gewählt wird. Anschließend werde ich die aus diesen Szenarien ableitbaren Anforderungen erörtern.

1.2.1 Newsticker

Seit 1999 benutzen Nachrichten-Agenturen wie z.B. Reuters oder AP zur Datenübermittlung an ihre Nachrichten-Bezieher den XML-basierten Standard News Industry Text Format (NITF), welcher im Jahre 2000 durch das ebenfalls XML-basierte NewsML ersetzt wurde.

Ein News-Ticker-System stellt hierbei ein typisches Szenario für die von mir in dieser Arbeit vorgestellten Verfahren dar. Auf der einen Seite steht die Nachrichten-Agentur, die einen kontinuierlichen Strom an verschiedensten Nachrichten produziert. Auf der anderen Seite steht der Bezieher, der nur an einem Teil der produzierten Nachrichten interessiert ist (z.B. nur Börsen- oder Sportnachrichten, nur regionale Nachrichten). Damit der Bezieher nicht den kompletten Nachrichten-Strom eines News-Tickers empfangen muss, steht zwischen Nachrichten-Agentur und Bezieher der Nachrichten-Broker. Dieser kennt die Interessen des Beziehers, filtert die für ihn interessanten Nachrichten aus dem Datenstrom und leitet sie an den Bezieher weiter.

Da in solch einem System eine sehr hohe Menge von Daten kontinuierlich versendet wird, kann dieses Szenario in hohem Maße vom Einsatz eines Kompressions-Verfahrens profitieren. Würde man nun einen einfachen, generischen Kompressor (wie z.B. gzip) verwenden, bedeutete dies aus Sicht des Brokers, dass er die komprimierten Nachrichten zunächst vollständig dekomprimieren müsste, bevor er die für den Bezieher interessanten Nachrichten herausfiltern kann, um sie anschließend wieder zu komprimieren und an den Bezieher zu versenden. Durch die zusätzliche Dekompression und Kompression entsteht also ein nicht unerheblicher Rechen-Mehraufwand insbesondere auf Seiten des Nachrichten-Brokers.

Beim Einsatz von XML-spezifischen Verfahren, die diese Filterung und gegebenenfalls die Modifikation des XML-Datenstroms direkt auf den komprimierten Daten unterstützen, entfällt dieser Mehraufwand, das gesamte System profitiert von der Kompression, ohne dass weitere Nachteile entstehen.

1.2.2 Daten-Management für mobile, Ajax-basierte Web 2.0 Anwendungen

Ajax [47] ist eine Programmier-Technik für interaktive Web-Anwendungen, die XML-Daten-Repräsentation in Form eines DOM-Baumes – also einer Baum-Repräsentation der hierarchischen XML-Daten – auf einem Client mit XMLHttpRequests als Daten-Austausch-Protokoll und JavaScript als Client-seitiger Programmiersprache kombiniert. Ajax erlaubt z.B., dass bei Benutzeraktionen synchron oder asynchron Teile des DOM-Baumes von einem Server nachgeladen werden, so dass der Benutzer eine schnelle Antwort auf seine Aktion erhält, ohne dass der komplette Dokument-Inhalt neu aufgebaut werden muss. Ein Beispiel für solche Anwendungen ist die automatische Vervollständigung des Suchbegriffs bei der Suche in einem Web-Lexikon. Das Potential, welches Ajax bietet, um interaktive Web-Anwendungen zu generieren, und der gegenwärtige Stand der Technik von Ajax werden in [70] dargestellt.

Um zur Laufzeit dynamische Modifikationen an Teilen einer im Client-Browser dargestellten Web-Anwendung vorzunehmen, benötigt ein XML-Kompressions-Verfahren, welches im Zusammenhang mit Ajax eingesetzt wird, die volle DOM-Unterstützung. Dies umfasst die Navigation (mindestens entlang der Achsen first-child, next-sibling und parent) sowie Updates (mindestens insert und remove) auf der komprimierten Darstellung.

Ersetzt man also die DOM-Komponente auf Client-Seite durch eine Navigations- und Update-fähige, komprimierte XML-Repräsentation, so können die restlichen Ajax-Komponenten unverändert übernommen werden. Statt unkomprimiertem XML wird nun komprimiertes XML übertragen, so dass Übertragungskosten eingespart werden. Die Darstellung der komprimierten XML-Repräsentation erfordert im Hauptspeicher deutlich weniger Speicher-Bedarf als die Darstellung des eigentlichen DOM-Baumes bei gleichem Funktionsumfang, so dass Arbeitsspeicher eingespart werden kann. Dadurch können bei gleichem Arbeitsspeicher deutlich umfangreichere Ajax-Anwendungen umgesetzt werden, und somit wird es auch mobilen Kleinstgeräten (wie z.B. Mobiltelefonen und PDAs) ermöglicht, Ajax-basierte Web 2.0 Anwendungen zu nutzen.

Teile der Ideen eines durch XML-Kompression verbesserten Daten-Managements für mobile, Ajax-basierte Web 2.0 Anwendungen wurden in [21] veröffentlicht.

1.2.3 Verbesserung der Cache-Kapazität durch Kompression

In herkömmlichen Caching-Szenarien speichert der Client in seinem Cache die Antworten auf zuvor gestellte Anfragen. Bei einer erneuten Anfrage muss nicht mehr das komplette Anfrage-Ergebnis vom Server an den Client gesendet wer-

den, sondern nur noch diejenigen Fragmente, die nicht im Cache enthalten sind. Dadurch müssen weniger Daten vom Server zum Client übertragen werden, so dass sowohl Transferkosten als eventuell auch Transferzeiten gesenkt werden.

Um in diesem Szenario allerdings die Antwort-Fragmente korrekt in den vorhandenen Cache-Inhalt zu integrieren, muss zusätzlich zu den eigentlichen Inhalten eines Fragmentes eine Identifizierungsinformation (wie z.B. eine Ordpath-Nummer [69]) mitgesendet werden. Insbesondere bei kleinen Antwort-Fragmenten ergibt sich so ein nicht unerheblicher Overhead.

Ebenso hat bei einem solchen Szenario im Allgemeinen der Client das Problem zu entscheiden, ob er alle benötigten Daten bereits in seinem Cache vorhanden hat. Dies ist entweder durch einen Teilmengen-Test möglich, in dem der Client testet, ob die zur Auswertung benötigten Fragmente eine Teilmenge der Daten sind, die von einer früheren Anfrage noch im Cache gespeichert sind. Solch ein Teilmengen-Test ist zwar für einige Teilklassen in Polynomzeit berechenbar ([18,19,40,62,64,77]), aber bereits für Anfragen, die gleichzeitig descendant-Achsen und Wildcards enthalten, NP-vollständig ([40,62,64]). Eine andere Möglichkeit sind sogenannte *Compensation-Anfragen* C, welche eine Umformung der ursprünglichen Anfrage A darstellen, so dass C angewandt auf den Cache genau dasselbe Ergebnis liefert wie A angewandt auf das Original-Dokument. Doch auch die Berechnung solch einer Compensation-Anfrage ist NP-vollständig, wie in [58,61] gezeigt wurde.

Bietet ein XML-Kompressions-Verfahren eine stark komprimierte Darstellung der XML-Struktur, ergibt sich ein neues Caching-Szenario: Bereits ab der Initialisierung der Anwendung enthält der Client-Cache die komplette Struktur des XML-Dokumentes (also alle öffenenden und schließenden XML-Tags inklusive der Attribut-Namen jedoch ohne Attribut- und Text-Werte). Als Antwort auf eine Anfrage muss der Server nur noch eine komprimierte Liste der noch nicht im Cache enthaltenen, zur Beantwortung der Anfrage notwendigen Konstanten in Dokumentreihenfolge übertragen (also die notwendigen Attribut- bzw. Text-Werte). Verwendet der Client ein Anfrage-Auswertungsverfahren, welches alle Knoten in Dokumentreihenfolge abarbeitet (wie z.B. das in Kapitel 9 vorgestellte Verfahren), so wird keinerlei zusätzliche Information zur korrekten Integration der nachgeladenen Daten in den Cache benötigt, diese ist durch die Übertragungsreihenfolge implizit gegeben.

Neben dem fehlenden Overhead aufgrund nicht benötigter Identifizierungsinformationen ist ein weiterer erheblicher Vorteil dieses neuen Szenarios, dass der Client nur aufgrund seines Cache-Inhaltes in linearer Zeit der Cache-Größe entscheiden kann, ob er bereits alle benötigten Informationen zur Beantwortung einer gegebenen Anfrage enthält. Der Client muss lediglich die Anfrage-Auswertung auf seinem Cache starten. Stößt er dabei nicht auf Text-Platzhalter, zu denen der Text-Wert noch nicht übertragen wurde, so enthält

der Cache bereits die vollständige Antwort, in diesem Fall muss keinerlei Datentransfer zwischen Server und Client stattfinden.

Teile der Ideen eines durch Struktur-Kompression verbesserten Cachings wurden in [14] zur Veröffentlichung eingereicht.

1.3 Anforderungen

Aus den eben vorgestellten Szenarien lassen sich Anforderungen ableiten, die erfüllt werden müssen, damit ein Kompressions-Verfahren besonders gut für diese Szenarien geeignet ist. In welchem Maße diese Anforderungen jedoch gewichtet sind, hängt in hohem Maße vom konkreten Szenario ab. Im folgenden gliedern sich die Anforderungen auf in allgemeine Anforderungen an die Kompressionsrate sowie Kompressions- und Dekompressionszeit, in Anforderungen an die Kompression von XML-Datenströmen und in Anforderungen an die Navigation bzw. Anfrage-Auswertung direkt auf der komprimierten Repräsentation. Abschließend werden diese Anforderungen noch einmal zusammenfassend aufgelistet.

1.3.1 Kompression und deren spezielle Anforderungen

Zunächst einmal muss ein Kompressions-Verfahren korrekt sein, also Kompression und Dekompression müssen Umkehroperationen voneinander darstellen. Dies bedeutet, dass man wieder das ursprüngliche XML-Dokument xml enthält, wenn man ein XML-Dokument xml komprimiert und anschließend mit dem entsprechenden Dekompressions-Verfahren wieder dekomprimiert.

Um einen Vorteil gegenüber anderen bereits verfügbaren XML-Kompressions-Verfahren zu erzielen, sollte ein neues Verfahren gegenüber anderen Verfahren, welche dieselben Anforderungen erfüllen, eine stärkere Kompressionsrate vorweisen. Hat das Kompressions-Verfahren hingegen starke Vorteile in Bezug auf andere Anforderungen, so kann ein gewisser Verlust bezüglich der Kompressionsrate in Kauf genommen werden. Ein Verfahren, welches z.B. weder Anfrage-Auswertung noch Updates direkt auf dem Komprimat erlaubt, wird voraussichtlich stärkere Kompressionsraten erreichen können als Verfahren, welche dies auf dem Komprimat zulassen. Da jedoch – wie im News-Ticker-Szenario beschrieben – durch die Anfrage-Auswertung direkt auf dem Komprimat ein Rechen-Mehraufwand vermieden werden kann, kann für viele Anwendungen ein gewisser Verlust der Kompressionsrate in Kauf genommen werden.

Da man in vielen Szenarien (z.B. News-Ticker) die komprimierten Daten bereits während des Kompressions-Vorgangs versenden möchte, sollten Kompression und Dekompression vergleichbar hohe Durchsätze erreichen, wie sie durch heute übliche Übertragungstechniken (z.B. ADSL) ermöglicht werden.

Da man jedoch typischerweise ein Dokument nur einmalig komprimiert, jedoch mehrfach und auf verschiedenen Rechnern dieses komprimierte Dokument weiterverarbeitet, indem man dieses z.B. dekomprimiert, ist ein etwas erhöhter Rechenaufwand bei der Kompression eher in Kauf zu nehmen als bei der Dekompression. Insbesondere ist es wünschenswert, dass mindestens genauso schnell dekomprimiert wie komprimiert werden kann, um einen möglichen Puffer-Überlauf beim Dekompressor und somit Empfänger der komprimierten Daten zu vermeiden.

1.3.2 XML-Datenströme und deren spezielle Anforderungen

Wann immer die Größe der Dokument-Repräsentation die Größe des verfügbaren Hauptspeichers überschreitet, erhalten wir besondere Anforderungen an die verarbeitenden Anwendungen – sowohl an die Kompression, als z.B. auch an die Anfrage-Auswertung. Dies ist insbesondere der Fall, wenn es um Datenverarbeitung auf mobilen Kleinstgeräten (wie z.B. Mobiltelefonen oder PDAs) geht. Besonders stark kommt dieser Aspekt allerdings zum Tragen, wenn es um die Verarbeitung von unendlichen Datenströmen, wie z.B. Datenströmen aus XML-News-Tickern geht. Dies führt zu einer Reihe von neuen Anforderungen.

Da die XML-Datenströme potentiell unendlich lang sind, ist nur ein einmaliger Durchlauf des Dokuments möglich. Es kann – wenn überhaupt – nur innerhalb eines gewissen Fensters vor und zurück navigiert werden. Da die Verarbeitung mindestens genauso schnell sein muss, wie die übertragenen Daten eintreffen, empfiehlt es sich oftmals, lediglich vorwärts, also linear, durch den Strom zu navigieren.

Zu einem Zeitpunkt ist immer nur ein kleiner Ausschnitt des Stroms bekannt. Aus diesem müssen dann die relevanten Daten herausgefiltert werden und eventuell für eine spätere Weiterverarbeitung zwischengespeichert werden. Hierbei ist es egal, ob ein Kompressions-Verfahren zur Verarbeitung von unendlichen Datenströmen diskrete oder gleitende Fenster verwendet, innerhalb derer das Verfahren unbegrenzt auf die Daten zugreifen kann. Bei gleitenden Fenstern wird das Fenster kontinuierlich weiterbewegt, das Fenster 'gleitet' also langsam während des Fortschreitens im XML-Datenstrom mit, und es existiert eine große Überschneidung des aktuellen Fensterinhalts mit dem vorherigen Fensterinhalt. Verfahren mit diskreten Fenstern bewegen das Fenster diskret weiter: Erst wird ein Teil des Stroms eingelesen und verarbeitet, anschließend ein weiterer. Die einzelnen Fensterinhalte haben in diesem Fall wenig bis gar keine Überschneidungen.

1.3.3 Anfrage-Auswertung und deren spezielle Anforderungen

Ein Verfahren, welches Navigation beziehungsweise Anfrage-Auswertung direkt auf dem Komprimat erlaubt, erfordert je nach Anwendung, dass das Ergebnis in komprimierter Form zur Verarbeitung weitergeleitet wird, oder aber dass das Ergebnis unabhängig vom Rest des Dokumentes dekomprimiert werden kann. Ein Kompressions-Verfahren muss also zunächst einmal partielle Dekompression erlauben, es muss also erlauben, dass Teile des Dokumentes dekomprimiert werden können, ohne dabei das gesamte Komprimat zu lesen. Die Unterstützung partieller Dekompression ist notwendig, um gegebenenfalls das Ergebnis einer Anfrage zur Weiterverarbeitung zu dekomprimieren.

Sowohl Navigation auf XML-Dokumenten wie auch Auswertung von XPath-Anfragen kann laut [50] auf wenige Basis-Operationen zurückgeführt werden. Diese Basis-Operationen umfassen die Navigation zum *first-child* – also zum ersten Kind-Knoten im XML-Baum – die Navigation zum *next-sibling* – also zum nachfolgenden Geschwister-Knoten im XML-Baum – die Navigation zum *parent* – also zum Elternknoten im XML-Baum – sowie die Ermittlung des Typs und des Labels eines Knotens. Ein Kompressions-Verfahren, welches Navigation und Anfrage-Auswertung direkt auf dem Komprimat unterstützt, muss somit also zunächst einmal zu jedem Knoten des ursprünglichen XML-Baums eine entsprechende Repräsentation im Komprimat bieten, welches diesen Knoten eindeutig identifiziert. Für all diese Knoten-Repräsentationen müssen dann die genannten Basis-Operationen unterstützt werden.

Manche Kompressions-Verfahren erlauben neben der Unterstützung der Basis-Operationen zur Navigation und zur Anfrage-Auswertung eine direkte, optimierte Unterstützung aller oder einiger XPath-Achsen. Ist eine solche Optimierung verfügbar, ist es natürlich sinnvoll diese Optimierung anzuwenden, um Navigation und Anfrage-Auswertung effizienter zu gestalten. Insgesamt ist es wünschenswert, dass Navigation und Anfrage-Auswertung mit vergleichbarem Rechenaufwand und Rechenzeit auf dem Komprimat durchgeführt werden können wie auf unkomprimiertem XML. Hat eine komprimierte XML-Repräsentation jedoch erhebliche andere Vorteile – wie z.B. eine sehr starke Kompressionsrate – so kann in einigen Anwendungen auch ein etwas höherer Rechenaufwand oder eine etwas höhere Rechenzeit in Kauf genommen werden.

1.3.4 Updates und deren spezielle Anforderungen

Szenarien wie komprimierte, Ajax-basierte Web-Anwendungen und Caching erfordern – neben der Navigation bzw. der Anfrage-Auswertung direkt auf dem Komprimat – zusätzlich die Möglichkeit, Updates direkt auf dem Komprimat zu unterstützen.

Ähnlich wie die Anfrage-Auswertung auf die oben genannten Basis-Operationen zurückgeführt werden kann, kann man auch Updates auf die Basis-Operationen *insert(XML xml, XML xmlIns, int p)*, welche den XML-Teilbaum xmlIns im XML-Dokument xml an Position p einfügt und *remove(XML xml, int p)*, welche den Teilbaum an Position p aus dem XML-Dokument xml entfernt, zurückführen.

Sollte dies in einem XML-Kompressions-Verfahren zu einer effizienteren Umsetzung führen, so können weitere Update-Operationen wie z.B. *replace(XML xml, XML xmlRep, Position p)*, welche den an Position p im XML-Dokument xml stehenden Teilbaum durch den XML-Teilbaum xmlRep ersetzt, direkt auf dem Komprimat umgesetzt werden, ohne den Umweg über die Hintereinanderausführung der Operationen remove und insert zu gehen.

Hinsichtlich der Unterstützung der Updates lassen sich zwei Qualitätsstufen unterscheiden: Einige Verfahren erzeugen durch Updates ein *korrektes*, nicht aber *optimales* Komprimat, während andere sowohl *korrekte* als auch *optimale* Komprimate erzeugen. Ein Komprimat ist hierbei *korrekt*, wenn die Hintereinanderausführung von Kompression, Update auf dem Komprimat und Dekompression zu demselben XML-Dokument xml' führt, wie die Durchführung des selben Updates direkt auf dem ursprünglichen XML-Dokument xml. Gilt zusätzlich, dass Kompression und Update auf dem Komprimat zu demselben Komprimat kxml' führen, wie Update auf dem XML-Dokument und anschließende Kompression, so ist dieses Update zusätzlich *optimal*. Dies bedeutet, dass nicht-optimale Updates im Allgemeinen zu einer Verschlechterung der Kompressionsrate führen, dennoch bieten Kompressions-Verfahren, welche bei Updates zu nicht-optimalen Komprimaten führen, einen erheblichen Vorteil gegenüber Kompressions-Verfahren, die keinerlei Updates direkt auf dem Komprimat erlauben, da im Falle von Updates der erhebliche Mehraufwand durch Dekompression und erneute Kompression eingespart werden kann.

1.3.5 Zusammenfassung der Anforderungen

Zusammenfassend lassen sich also die folgenden Anforderungen nennen, welche durch navigierbare Kompressions-Verfahren erfüllt werden sollten, so dass z.B. die oben genannten Szenarien in hohem Maße von der Verwendung dieser Verfahren profitieren können:

- **Anforderung 1:** Kompression und Dekompression müssen zueinander invers sein, die Dekompression der komprimierten Repräsentation muss also bei Eingabe eines beliebigen validen Dokuments wieder das ursprüngliche Dokument erzeugen.

- **Anforderung 2:** Die Kompressionsrate muss mindestens so stark sein wie die anderer XML-Kompressions-Verfahren mit vergleichbaren Eigenschaften.
- **Anforderung 3:** Kompression und Dekompression müssen vergleichbare Durchsätze erreichen wie derzeit übliche Übertragungsverfahren (z.B. ADSL).
- **Anforderung 4:** Die Dekompression muss mindestens so schnell sein wie die Kompression.
- **Anforderung 5:** Kompression und Dekompression müssen möglich sein, ohne dass das gesamte Dokument beziehungsweise das gesamte Komprimat bekannt ist.
- **Anforderung 6:** Zu jedem Knoten des ursprünglichen XML-Dokumentes muss eine eindeutige Repräsentation im Komprimat existieren.
- **Anforderung 7:** Partielle Dekompression, also Dekompression von XML-Teilbäumen innerhalb des Komprimats, muss möglich sein.
- **Anforderung 8:** Die Basis-Operationen first-child, next-sibling, parent sowie die Ermittlung des Typs und des Labels eines Knotens direkt auf dem Komprimat müssen unterstützt werden.
- **Anforderung 9:** Die Anfrage-Auswertungszeiten auf dem Komprimat sollten hierbei vergleichbar zu Anfrage-Auswertungszeiten auf unkomprimiertem XML sein.
- **Anforderung 10:** Die Basis-Operationen insert und remove müssen direkt auf dem Komprimat unterstützt werden.

Zum Zeitpunkt der Veröffentlichung der in dieser Arbeit vorgestellten Verfahren war noch kein Verfahren bekannt, welches all diese Anforderungen in dem Maße unterstützt, wie es die in dieser Arbeit vorgestellten Verfahren tun.

1.4 Beitrag dieser Arbeit und Gesamtüberblick

In dieser Arbeit leiste ich den folgenden Beitrag zur Erfüllung der im vorigen Abschnitt präsentierten Anforderungen:

- Trennung einer XML-Repräsentation in eine Struktur-Repräsentation, welche die inneren Knoten eines XML-Baums enthält und eine Konstanten-Repräsentation, welche die Blattknoten eines XML-Baums enthält.
- Entwicklung und Definition dreier neuer, komprimierter XML-Struktur-Kompressions-Verfahren, wobei jedes dieser Verfahren die oben genannten Anforderungen erfüllt. Jedes Verfahren erreicht jedoch ein unterschiedlich hohes Maß an Erfüllung der Anforderungen, so dass jedes

Verfahren andere Stärken hat und somit entsprechend den in der jeweiligen Anwendung erforderlichen Anforderungen das beste Verfahren ausgewählt werden kann. Zusammenfassend werden für jedes dieser drei Verfahren

  - die Korrektheit bewiesen, es wird also bewiesen, dass die Dekompression des Komprimats wieder das ursprüngliche XML-Dokument erzeugt.
  - die Korrektheit der Navigation auf dem Komprimat mit Hilfe der Basis-Operationen first-child, next-sibling, parent, getLabel und getType nachgewiesen.
  - Update-Möglichkeiten direkt auf dem Komprimat und Unterstützung der DOM-Schnittstelle durch das Kompressions-Verfahren erörtert.
  - die Verarbeitung von unendlichen Datenströmen dargestellt.

- Vorstellung zweier Kombinationsmöglichkeiten dieser Verfahren, so dass auch diese beiden hybriden Verfahren die oben genannten Anforderungen erfüllen.
- Vorstellung und Vergleich verschiedener Verfahren zur Konstanten-Kompression sowie deren Integration mit der Repräsentation der XML-Struktur.
- Entwicklung und Definition eines XPath-Auswertungs-Verfahrens, welches XPath-Anfragen auf allen XML-Repräsentationen, die die Basis-Operationen first-child, next-sibling, parent, getLabel und getType unterstützen, auswertet. Die Auswertungszeit dieses Verfahrens ist hierbei linear, also proportional zur Dokument-Größe.
- Vergleich der vorgestellten Verfahren bezüglich ihrer Kompressionsrate sowie ihrer Kompressions-, Dekompressions- und Anfrage-Auswertungs-Zeiten untereinander sowie mit anderen XML-Kompressions-Verfahren.

1.5 Gliederung dieser Arbeit

Diese Arbeit gliedert sich wie folgt:

- Kapitel 2 definiert die theoretischen Grundlagen von XML, SAX, XPath und DOM.
- Kapitel 3 definiert verschiedene SAX-Strom-Varianten, die als Eingabe für die verschiedenen vorgestellten Verfahren genutzt werden.

- Kapitel 4, 5 und 6 stellen den Hauptteil dieser Arbeit dar. Sie beschreiben die drei Struktur-Kompressions-Verfahren *Succinct*(Kodierungs-basierte Kompression), *DAG*(Kompression basierend auf Struktur-Redundanzen) und *DTD-Subtraktion*(Schema-basierte Kompression).
- Kapitel 7 stellt eine Kombination von auf Struktur-Redundanzen basierten Verfahren mit Kodierungs-basierter bzw. mit Schema-basierter XML-Kompression vor.
- Kapitel 8 beschreibt die Integration der komprimierten Daten mit der komprimierten XML-Struktur-Repräsentation und stellt einige Verfahren zur Daten-Kompression vor.
- Kapitel 9 beschreibt das Verfahren zur linearen Anfrage-Auswertung auf navigierbaren XML-Repräsentationen.
- Kapitel 10 vergleicht die in dieser Arbeit präsentierten Ideen mit den bereits in der Literatur existierenden.
- Kapitel 11 enthält die Performanz-Auswertungen und Vergleiche der in den Kapiteln 4, 5, 6 und 7 vorgestellten Verfahren bezüglich Kompressionsrate sowie bezüglich Kompressions-, Dekompressions- und Anfrage-Auswertungs-Zeiten untereinander sowie mit anderen verfügbaren XML-Kompressions-Verfahren.
- Kapitel 12 diskutiert, von welchen der in dieser Arbeit vorgestellten Verfahren die in diesem Kapitel vorgestellten Szenarien besonders stark profitieren.
- Kapitel 13 enthält eine Zusammenfassung dieser Arbeit sowie einen Ausblick auf weiterführende Forschungsansätze.

2 Grundlagen

2.1 XML

XML (Extensible Markup Language) [23] ist eine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdateien. Ein XML-Dokument besteht aus den folgenden fünf verschiedenen Komponenten:

- *Elemente*
  Diese werden entweder durch ein Paar aus Start-Tag (<Tag-Name>) und End-Tag (</Tag-Name>) oder durch einen Empty-Element-Tag (<Tag-Name/>) dargestellt.
- *Attribut-Wert-Paare*
  Diese sind Schlüsselwort-Wert-Paare (Attribut= "Attribut-Wert"), die in einem Start-Tag oder einem Empty-Element-Tag auf den Element-Namen folgen.
- *Verarbeitungsanweisungen*
  (<?Ziel-Name Parameter ?>).
- *Kommentare*
  (<!– Kommentar-Text –>).
- *Text*
  Text kann als normaler Text oder in Form eines CDATA-Abschnittes (<![CDATA[beliebiger Text]]>) auftreten.

Viele Anwendungen greifen jedoch nur auf die eigentlichen Inhalte zu, die durch Elemente, Attribute, Attribut-Werte und Texte dargestellt werden, während Kommentare und Verarbeitungsanweisungen ignoriert werden. Auch die sogenannten *Whitespaces*, also die Leerzeichen, Tabulatoren und Zeilenumbrüche, die zur Formatierung zwischen den Tags stehen, nicht jedoch einem eigentlichen Text-Element entsprechen, werden von den meisten Anwendungen ignoriert. Daher werde ich in den nachfolgenden Kapiteln alle vorgestellten Verfahren auf die Darstellung des Inhaltes, also auf die Elemente, Attribute, Attribut-Werte und Texte beschränken.

Durch die Schachtelung von Elementen ineinander entsteht eine streng hierarchische Struktur. Diese erlaubt es, beliebige XML-Dokumente als Baum darzustellen. Dazu werden jedem Element die direkt untergeordneten Elemente sowie die Attribute und die direkt eingeschlossenen Texte als Kindknoten zugeordnet. Formal kann man den Inhalt eines XML-Dokumentes mit Hilfe eines Baumes wie folgt definieren:

Definition 2.1 (XML-Baum)**.** Ein *XML-Baum* $xml = (V, E)$ sei ein geordneter Baum, wobei V die Menge der Knoten sei und $E \subseteq (V \times V)$ die Menge der Kanten. Die Menge V der Knoten teilt sich in die disjunkten Teilmengen $V = \{r\} \cup VE \cup VA \cup VAW \cup VT$ auf, wobei r der *Wurzelknoten*, VE die *Element-Menge*, VA die *Attribut-Menge*, VAW die *Menge der Attributwerte* und VT die *Text-Menge* ist. Für diese Mengen gilt

- $not\ \exists v \in V : (v, r) \in E$
- $\forall vt \in (VT \cup VAW) : not\ \exists v \in V : (vt, v) \in E$
- $\forall va \in VA\ \exists vaw \in VAW : (va, vaw) \in E.$

Zu einem Knoten $v \in V$ sei $v.label$ das Label von v. $\Sigma := \bigcup_{v \in V} v.label$ sei die *Menge aller Label* von xml.

Im weiteren Verlauf bezeichne $S = SV \cup SE$ mit $SV = \{r\} \cup VE \cup VA$ und $SE = \{(a, b) \in E | a, b \in SV\}$ den Strukturanteil eines XML-Baumes und $D = VT \cup VAW$ den Daten- bzw. Konstantenanteil. □

Bei der Verarbeitung von XML hat ein herkömmlicher XML-Baum den Nachteil, dass jeder Knoten beliebig viele Kindknoten haben kann. Im Gegensatz dazu ist die Binärbaum-Darstellung eines XML-Dokuments einfacher zu verarbeiten. Hierbei hat jeder Knoten maximal zwei Zeiger auf Nachfolgerknoten. Der erste Zeiger – der first-child-Zeiger – zeigt auf den ersten Kindknoten, während der zweite Zeiger – der next-sibling-Zeiger – auf den nächsten Geschwisterknoten zeigt, also auf denjenigen Knoten, der den selben Elternknoten hat, und in Reihenfolge der Breitensuche direkt auf den aktuellen Knoten folgt.

Beide Darstellungen – der herkömmliche XML-Baum und der binäre XML-Baum – haben exakt die selbe Anzahl an Kanten und Knoten.

2.2 DTD

Eine DTD (Document Type Definition) [23] ist eine Menge von Produktionsregeln, welche eine Menge von gültigen XML-Dokumenten bzw. XML-Bäumen definiert. Eine DTD besteht aus einer Liste von Elementtyp-Deklarationen, Attributlisten-Deklarationen, Entity-Deklarationen und Notation-Deklarationen, die wie folgt aufgebaut sind:

Eine Elementtyp-Deklaration legt die Definition eines Elementes e sowie die Beziehungen zwischen dessen Kindern untereinander fest. Sie wird entsprechend der folgenden Grammatik gebildet:

```
elementdecl  ::=  '<!ELEMENT ' Name ' ' contentspec '>'
contentspec  ::=  'EMPTY' | 'ANY' | c
c            ::=  c'*' | c'+' | c'?' |
                  Name | 'PCDATA' | '('c','c') | '('c'|'c')'
```

Hierbei haben die Operatoren 'EMPTY', 'ANY', 'PCDATA', ',', '|' '*', '+', '?' die folgende Bedeutung:

- *EMPTY*
  Das definierte Element hat keinen Inhalt, stellt also einen Blattknoten im XML-Baum dar.
- *ANY*
  Das definierte Element hat beliebigen Inhalt, der Inhalt ist also in dieser DTD nicht näher spezifiziert.
- *PCDATA*
  Das definierte Element hat als Inhalt einen Text-Wert.
- *a,b (Sequenz)*
  Auf die durch a definierte Liste von Kindknoten folgt die durch b definierte Liste von Kindknoten.
- *a|b (Choice)*
  Es folgt entweder die durch a definierte Liste von Kindknoten oder die durch b definierte Liste von Kindknoten.
- *a* (Kleene)*
  Die durch a definierte Liste von Kindknoten kommt beliebig oft vor.
- *a+ (Plus)*
  Die durch a definierte Liste von Kindknoten kommt beliebig oft vor, mindestens ist sie jedoch einmal vorhanden.
- *a? (Option)*
  Die durch a definierte Liste von Kindknoten ist optional, ist also einmal oder keinmal vorhanden.

Eine Attributlisten-Deklaration legt eine Liste von Attributen zu einem Element fest. Sie hat die Form *<!ATTLIST Elementname Attributliste>*, wobei Elementname der Name eines Elementes ist, und Attributliste eine Liste von Attributen ist, in der für jedes Attribut Attributname, Attributtyp sowie Attributvorgaben festgelegt werden. Attributtyp kann hierbei 'CDATA', 'ID', 'IDREF', 'IDREFS', 'NMTOKEN', 'NMTOKENS', 'ENTITY', 'ENTITIES', 'NOTATION', 'NOTATIONS', sowie Aufzählungen und NOTATION-

Aufzählungen sein. Eine Attributvorgabe ist entweder '#REQUIRED' (das Attribut muss angegeben werden), '#IMPLIED' (das Attribut ist optional), "Wert" (dieser Wert gilt, falls das Attribut nicht angegeben wird) oder '#FIXED "Wert"' (dieser Wert ist immer der Attributwert).

Da die Entity-Deklarationen und die Notation-Deklarationen für die in dieser Arbeit vorgestellten Kompressionsverfahren keine weitere Bedeutung haben, wird an dieser Stelle nicht näher auf diese eingegangen.

Neben einer DTD gibt es noch andere Möglichkeiten zur Spezifikation des XML-Inhaltsmodelles wie z.B. XML Schema [42] oder RelaxNG [36]. Da die in dieser Arbeit beschriebenen Verfahren auf der DTD als Inhaltsmodell basieren, werde ich an dieser Stelle nicht näher auf die alternativen Inhaltsmodelle eingehen.

2.3 SAX

Das *Simple API for XML (SAX)* ist eine unabhängige Programmier-Schnittstelle, die es erlaubt, XML-Dokumente sequentiell zu verarbeiten. Ein Parser durchläuft das XML-Dokument und erzeugt entsprechend des gelesenen Inputs eine Folge aus SAX-Events, wobei jedes SAX-Event aus der Menge der folgenden SAX-Events ist(die hier vorgestellten Events weichen zwecks einer einfacheren Darstellung bzgl. der Attribute von der üblichen SAX-Darstellung ab. In der hier vorgestellten Darstellung werden für Attribute eigenständige Events erzeugt, wohingegen diese in der üblichen SAX-Schnittstelle Bestandteil der Element-Events sind):

- *startDocument()*
  Das SAX-Event startDocument wird einmalig zu Beginn eines Durchlaufs des XML-Dokumentes bzw. des XML-Stroms erzeugt.
- *startElement(String name)*
  Das SAX-Event startElement wird für jeden gelesenen öffnenden Tag erzeugt. Hierbei enthält der Parameter *name* den Elementnamen.
- *startAttribute(String name)*
  Das SAX-Event startElement wird für jedes gelesene Attribut erzeugt. Hierbei enthält der Parameter *name* den Attributnamen.
- *characters(String text)*
  Das SAX-Event characters wird für alle gelesenen Text-Elemente erzeugt, also für alle Texte und Attributwerte. Der Parameter *text* enthält hierbei die gelesenen Zeichen.
- *endAttribute(String name)*
  Das SAX-Event endElement wird für jedes gelesene Attribut nach Verar-

beitung des Attributwertes erzeugt. Hierbei enthält der Parameter *name* den Attributnamen.

- *endElement(String name)*
  Das SAX-Event endElement wird für jeden gelesenen schließenden Tag erzeugt. Hierbei enthält der Parameter *name* den Elementnamen.
- *endDocument()*
  Das SAX-Event endDocument wird einmalig am Ende eines Durchlaufs des XML-Dokuments bzw. des XML-Stroms erzeugt.

Durch die sequentielle Verarbeitung benötigen Programme, die auf SAX basieren, wenig Arbeitsspeicher im Vergleich zur Dokumentgröße. Daher sind SAX-basierte Programme insbesondere für unbegrenzte XML-Datenströme oder für mobile Endgeräte mit wenig Arbeitsspeicher geeignet.

Formal kann man einen SAX-Strom zu einem gegebenen XML-Baum wie folgt definieren. Hierbei bezeichnet der Operator $\otimes$ die Konkatenation zweier Listen.

Definition 2.2 (SAX-Event-Strom zu einem XML-Baum)**.** Sei xml ein XML-Baum entsprechend Definition 2.1 mit Knotenmenge $V = \{r\} \cup VE \cup VA \cup VAW \cup VT$ und Kantenmenge $E \subseteq (V \times V)$ und Labelmenge Σ. Sei Ev={startDocument()}$\cup$ {endDocument()}$\cup$ {startElement(σ)$|$ $\sigma \in \Sigma$} $\cup$ {endElement(σ)$|$ $\sigma \in \Sigma$} $\cup$ {startAttribute(σ)$|$ $\sigma \in \Sigma$} $\cup$ {endAttribute(σ)$|$ $\sigma \in \Sigma$}$\cup$ {characters(σ)$|$ $\sigma \in \Sigma$} die Menge aller SAX-Events.

Zu einem Knoten $v \in V$ mit Kindknoten $cv_1, \ldots, cv_n$ sei der SAX-Event-Strom $sax(v) : V \to Ev*$ definiert durch

$$sax(v) := \begin{cases} (startDocument()) \otimes \\ (startElement(v.label)) \otimes sax(cv_1) \otimes \\ \ldots \otimes sax(cv_n) \otimes (endElement(v.label)) \otimes \\ (endDocument()) & falls\ v = r \\ (startElement(v.label)) \otimes sax(cv_1) \otimes \\ \ldots \otimes sax(cv_n) \otimes (endElement(v.label)) & falls\ v \in VE \\ (startAttribute(v.label)) \otimes sax(cv_1) \otimes \\ (endAttribute(v.label)) & falls\ v \in VA \\ (characters(v.label)) & falls\ v \in VT \cup VAW \end{cases}$$

Der SAX-Event-Strom des XML-Baumes xml ist dann definiert durch $SAX(xml) := sax(r)$. □

Definition 2.3 (Position eines Knotens im SAX-Event-Strom)**.** Sei xml ein XML-Baum entsprechend Definition 2.1 mit Knotenmenge $V = \{r\} \cup VE \cup VA \cup VAW \cup VT$ und Kantenmenge $E \subseteq (V \times V)$ und Labelmenge Σ.

Sei $SAX(xml) = (s_1, \ldots, s_n)$ der SAX-Event-Strom zu xml. Sei weiterhin $v \in V$ ein Knoten des XML-Baums xml mit $sax(v) = (s_i, \ldots, s_j), 1 \leq i, j \leq n$ ist eine Teilfolge von $SAX(xml)$.

Dann ist i die *Position des Knotens v im SAX-Event-Strom zum XML-Baum* xml.

□

2.4 XPath

Die XML Path Language (XPath) [37] ist eine vom W3C-Konsortium entwickelte Anfragesprache, um mit Hilfe von Pfad-Anfragen Teile eines XML-Dokumentes zu adressieren.

Eine XPath-Anfrage ist eine Liste sogenannter *Location-Steps*, wobei jeder Location-Step aus einem Achsen-Test, einem Knoten-Test und einer Folge von Prädikat-Filtern besteht. Ein XML-Knoten x erfüllt einen Location-Step angewandt auf einen Kontextknoten k, wenn k und x den Achsen-Test erfüllen, wenn x.label den Knoten-Test erfüllt sowie wenn jeder der vorhandenen Prädikat-Filter zu true evaluiert werden kann. Das Ergebnis einer XPath-Anfrage XP, bestehend aus den Location-Steps $ls_1, \ldots, ls_n$, kann man berechnen, indem man ls_1 auf den Wurzelknoten r eines XML-Baumes anwendet, und ls_i für $1 < i \leq n$ auf die erfüllenden Knoten von ls_{i-1}. Das Ergebnis von XP angewandt auf xml sind dann die erfüllenden Knoten von ls_n.

Lassen wir die Unterscheidung zwischen Element-, Attribut- und Text-Knoten außen vor, so existieren 10 verschiedene XPath-Achsen, die jedoch alle auf die schon im binären XML-Baum auftretenden Achsen first-child und next-sibling, sowie die Achse self zurückführbar sind. Diese binären oder auch atomaren XPath-Achsen sind wie folgt definiert.

Definition 2.4 (first-child, next-sibling und self)**.** Sei xml ein XML-Dokument entsprechend Definition 2.1 mit Wurzel-Knoten r und sei sax(xml)= $(sr) \otimes \ldots \otimes (er)$ ein SAX-Event-Strom zu xml entsprechend Definition 2.2. Seien $E1, E2 \in VE$ zwei Element-Knoten in xml mit sax(E1)= $(sE1) \otimes \ldots \otimes (eE1)$ und sax(E2)= $(sE2) \otimes \ldots \otimes (eE2)$. Dann bezeichnen wir

(a) E2 als first-child von E1, E2=first-child(E1), genau dann, wenn sax(E1)= $(sE1) \otimes (sE2) \otimes \ldots \otimes (eE1)$.

(b) E2 als next-sibling von E1, E2=next-sibling(E2) genau dann, wenn sax(r)= $(sr) \otimes \ldots \otimes (sE1) \otimes \ldots \otimes (eE1) \otimes (sE2) \otimes \ldots \otimes (er)$

(c) E2 als self von E1, E2=self(E1) genau dann, wenn E1=E2.

□

Basierend auf der Definition der binären Achsen kann man nun gemäß einer Ideen von [50] alle XPath-Achsen wie folgt definieren:

Definition 2.5 (XPath-Achsen)**.** Sei xml ein XML-Baum entsprechend Definition 2.1 und $E1 \in V$ ein Knoten in xml. Dann bezeichnen wir

(a) die rekursiv definierte Element-Menge E1/child:={Ek | Ek=first-child(E1) ∨(∃ E2 ∈ V: E2 ∈ E1/child ∧ Ek=next-sibling(E2))} als child-Knoten von E1.

(b) das Element E1/parent:=(Ek | E1 ∈ Ek/child) als parent-Knoten von E1.

(c) die rekursiv definierte Element-Menge E1/descendant:={Ek | Ek ∈ child(E1) ∨(∃ E2 ∈ V: E2 ∈ E1/descendant ∧ Ek ∈ E2/child)} als descendant-Knoten von E1.

(d) die Element-Menge E1/descendant-or-self:={Ek | Ek=self(E1) ∨ Ek ∈ E1/descendant} als descendant-or-self-Knoten von E1.

(e) die Element-Menge E1/ancestor:={Ek | E1 ∈ Ek/descendant} als ancestor-Knoten von E1.

(f) die Element-Menge E1/ancestor-or-self:={Ek | E1 ∈ Ek/descendant-or-self} als ancestor-or-self-Knoten von E1.

(g) die Element-Menge E1/following-sibling:={Ek | Ek=next-sibling(E1) ∨(∃ E2 ∈ V: E2 ∈ E1/following-sibling ∧ Ek=next-sibling(E2))} als following-sibling-Knoten von E1.

(h) die Element-Menge E1/preceding-sibling:={Ek | E1 ∈ Ek/following-sibling} als preceding-sibling-Knoten von E1.

(i) die Element-Menge E1/following:={Ek | ∃ E2, E3 ∈ V: E2 ∈ E1/ancestor-or-self ∧ E3 ∈ E2/following-sibling ∧ Ek ∈ E3/descendant-or-self} als following-Knoten von E1.

(j) die Element-Menge E1/preceding:={Ek | E1 ∈ Ek/following} als preceding-Knoten von E1.

Hierbei werden die Achsen self, child, descendant, descendant-or-self, following-sibling und following als *Vorwärtsachsen* und die Achsen parent, ancestor, ancestor-or-self, preceding-sibling und preceding als *Rückwärtsachsen* bezeichnet. □

Formal können wir nun eine XPath-Anfrage wie folgt definieren:

Definition 2.6 (XPath-Anfrage)**.** Sei xml ein XML-Baum entsprechend Definition 2.1 mit der Menge Σ von Labeln. Sei $\Sigma' := \Sigma \cup \{'*'\}$. Sei x eine XPath-Achse entsprechend Definition 2.5. Dann definiert die folgende EBNF-Grammatik mit Start-Symbol cxp eine gültige XPath-Anfrage.

```
cxp           ::=   '/' locationpath
locationpath  ::=  locationstep ('/' locationstep)*
locationstep  ::=  x '::' Σ' | x '::' Σ' '[' pred ']'
pred          ::=  locationpath | locationpath comp constant |
                   '(' pred 'and' pred ')' |
                   '(' pred 'or' pred ')'
comp          ::=  '=' | '<' | '≤' | '>' | '≥' | '≠'
```

□

2.4.1 Eliminerung der Rückwärtsachsen und der Following-Achse

Gewisse Anwendungen – wie z.B. die Auswertung von unendlichen SAX-Strömen – erfordern eine strenge lineare Durchquerung des Stroms. Dies bedeutet insbesondere, dass man nicht innerhalb des Stroms zurückspringen kann, was die Auswertung der Rückwärtsachsen erschwert.

Zwar lassen sich – wie im vorhergehenden Abschnitt gezeigt – die Vorwärtsachsen (mit Ausnahme der Achse following) direkt auf die atomaren Achsen first-child und next-sibling zurückführen, so dass einer linearen Auswertung dieser Achsen nichts im Wege steht. Zur Auswertung der Rückwärtsachsen jedoch werden die Inversen der Vorwärtsachsen benötigt, was einer entgegengesetzten Navigation im Strom entspricht.

[68] bietet ein Verfahren an, das es erlaubt, beliebige Anfragen in äquivalente Anfragen umzuschreiben, die frei von Rückwärtsachsen sind. Soll also eine Anfrage nicht nur auf die atomaren Achsen abgebildet werden, sondern gleichzeitig von allen Rückwärtsachsen befreit werden, so bietet sich das folgende Verfahren an:

1. Eliminierung aller Rückwärtsachsen entsprechend des in [68] beschriebenen Verfahrens. Nach diesem Schritt erhalten wir eine Anfrage bestehend aus den Achsen descendant-or-self, descendant, self, child, following und following-sibling.
2. Umschreiben aller following-Achsen mit Hilfe der folgenden äquivalenzerhaltenden Umformungsregel:
   following → ancestor-or-self/following-sibling/descendant-or-self.
   Nach diesem Schritt erhalten wir eine Anfrage bestehend aus den Achsen descendant-or-self, descendant, self, child, following-sibling und ancestor-or-self.
3. Erneute Anwendung des in [68] beschriebenen Verfahrens, um die im zweiten Schritt entstandenen ancestor-or-self Achsen zu eliminieren. Nach Abschluss aller Schritte enthält die Anfrage nun nur noch die Achsen descendant-or-self, descendant, self, child und following-sibling.

4. Abbilden der verbleibenden Achsen auf first-child und next-sibling.

Dadurch erhalten wir eine reine Abbildung auf die Achsen first-child und next-sibling.

2.5 DOM

Das Document Object Model (DOM) [54] ist eine Programmier-Schnittstelle für sowohl lesenden als auch schreibenden Zugriff auf XML-Bäume. Sie wurde vom W3C-Konsortium definiert. Die DOM-Schnittstelle umfasst eine Vielzahl verschiedener Objekte (z.B. Document und Node) und darauf ausführbarer Methoden. Ähnlich wie man XML in eine äquivalente Binärbaum-Darstellung überführen kann und XPath auf Ausdrücke über die atomaren Achsen first-child, next-sibling und self normalisieren kann, so kann man auch die Funktionalitäten von DOM auf wenige Grundfunktionalitäten zurückführen.

Grundlegender Gedanke der DOM-Schnittstelle ist es, dass man das XML-Dokument als einen Baum betrachtet, der sich komplett im Hauptspeicher befindet, und innerhalb dessen beliebig über die XPath-Achsen navigiert werden kann, und der beliebig durch Einfügen, Löschen und Ändern manipuliert werden kann.

2.5.1 Lesende DOM-Operationen

Die Menge aller lesenden DOM-Operationen kann mit einer Rückführung dieser Operationen auf die Navigation mittels XPath-Ausdrücken im DOM-Baum realisiert werden. So liefert beispielsweise die DOM-Methode *Document.getElementById(String elementID)* alle Dokument-Knoten zurück, die ein Attribut "id" enthalten, wobei das Attribut "id" den Attribut-Wert *elementID* des Parameters hat. Diese kann z.B. mit Hilfe des XPath-Ausdrucks *//descendant-or-self::*[./attribute::id='elementID']* simuliert werden.

Da aber jede XPath-Anfrage auf die Achsen-Menge first-child, next-sibling, self und parent[1] zurückgeführt werden kann, genügt es also, dass eine XML-Repräsentation diese atomaren Achsen unterstützt, so dass die Menge aller lesenden DOM-Operationen auf diesem Verfahren ausgeführt werden können.

2.5.2 Schreibende DOM-Operationen

Hinweis: Im Folgenden wird auf die Position p innerhalb eines XML-Dokuments bzw. XML-Baums verwiesen. Diese sei analog zum SAX-Event-Strom wie in Definition 2.3 definiert.

[1] Da bei einer DOM-Anwendung der Kontextknoten nicht zwangsläufig der Wurzelknoten des XML-Dokumentes ist, muss zusätzlich noch die parent-Achse realisiert werden.

Ebenso wie die lesenden DOM-Operationen auf die Navigation via XPath zurückgeführt werden können, kann man die schreibenden DOM-Operationen auf folgende Schreiboperationen auf XML zurückführen:

- *insert(XML xml, XML xmlIns, int p)*
  Fügt den übergebenen XML-Teilbaum xmlIns im XML-Dokument xml an Position p ein.
- *remove(XML xml, int p)*
  Entfernt den Teilbaum an Position p aus dem XML-Dokument xml.
- *replace(XML xml, XML xmlRep, Position p)*
  Ersetzt den an Position p im XML-Dokument xml stehenden Teilbaum durch den XML-Teilbaum xmlRep.

Da die Methode *replace* durch ein Hintereinanderausführen der Methoden *remove* und *insert* simuliert werden kann, werde ich für alle XML-Repräsentationen die schreibenden Operationen *insert* und *remove* definieren, um zu motivieren, dass es möglich ist, eine DOM-Schnittstelle für diese Repräsentationen zu implementieren.

Die Operation *insert(XML xml, XML xmlIns, int p)* fügt ein XML-Dokument xmlIns an gegebener Position p in das XML-Dokument xml ein. Algorithmus 2.1 zeigt eine Umsetzung dieser Operation, wobei die Operation *concat(XML xml1, XML xml2)* die Konkatenation zweier XML-Fragmente berechnet, und die Operation *subsequence(XML xml, int start, int end)* ein Teilfragment von XML berechnet, das an Position start beginnt und an Position end endet.

```
1 public XML insert(XML xml, XML xmlIns, int p){
2     XML xmlNeu = subsequence(xml, 1, p-1);
3     xmlNeu = concat(xmlNeu, xmlIns);
4     xmlNeu = concat(xmlNeu, subsequence(xml, p, xml.
          length));
5     return xmlNeu;
6 }
```

Algorithmus 2.1: insert-Operation für XML-Dokumente

In Definition 2.7 wird die Funktion $insert_{xml,xml_{ins},p}(x)$ definiert, die den Zusammenhang zwischen den XML-Dokumenten xml und xml_{ins} vor und dem XML-Dokument xml_{neu} nach der Operation herstellt. Sie wird später benötigt, um die Korrektheit der Update-Operationen auf den XML-Repräsentationen nachzuweisen.

Definition 2.7 (insert)**.** Seien $xml = (xml_1, \ldots, xml_n)$ und $xml_{ins} = (xml_{ins_1}, \ldots, xml_{ins_m})$ zwei XML-Dokumente. Sei p eine Position in xml mit 1≤p≤n+1. Dann ist die Funktion $insert_{xml,xml_{ins},p}(x)$: {1, ..., n+m} → XML definiert als:

$$insert_{xml,xml_{ins},p}(x) := \begin{cases} xml_x & falls\ x < p \\ xml_{ins_{x-p+1}} & falls\ p \leq x < p+m \\ xml_{x-m} & sonst \end{cases}$$

$xml_{neu} := (insert_{xml,xml_{ins},p}(1), \ldots, insert_{xml,xml_{ins},p}(n+m))$ □

Sei p die Position eines startElement-Events in xml. Die Operation *remove(XML xml, int p)* löscht einen Teilbaum beginnend an gegebener Position p aus einem XML-Dokument xml. Algorithmus 2.2 zeigt eine Umsetzung dieser Operation, wobei die Operation *endTag(XML xml, int pos)* die Position des End-Tags zu einem durch die Position pos gegeben Start-Tag berechnet.

```
1 public XML remove(XML xml, int p){
2    XML xmlNeu = subsequence(xml, 1, p-1);
3    xmlNeu = concat(xmlNeu, subsequence(xml, endTag(xml,p
         )+1, xml.length));
4    return xmlNeu;
5 }
```

Algorithmus 2.2: remove-Operation für XML-Dokumente

In Definition 2.8 wird die Funktion $remove_{xml,p}(x)$ definiert, die den Zusammenhang zwischen dem XML-Dokument xml vor und dem XML-Dokument xml_{neu} nach der Operation herstellt. Sie wird später benötigt, um die Korrektheit der Update-Operationen auf den XML-Repräsentationen nachzuweisen.

Definition 2.8 (remove)**.** Sei $xml = (xml_1, \ldots, xml_n)$ ein XML-Dokument. Sei xml_p der Start-Tag eines XML-Elements E mit Position p und sei xml_k der End-Tag von E mit Position k im XML-Dokument xml. Sei l:=k-p. Dann ist die Funktion $remove_{xml,p}(x)$: {1, ..., n-l} → XML definiert als:

$$remove_{xml,p}(x) := \begin{cases} xml_x & falls\ x < p \\ xml_{x+l} & sonst \end{cases}$$

$xml_{neu} := (remove_{xml,p}(1), \ldots, remove_{xml,p}(n-l))$ □

2.6 Speichereffiziente Darstellung von ganzzahligen natürlichen Werten

In einigen der vorgestellten Verfahren zur XML-Kompression wird ein Verfahren benötigt, um ganzzahlige Werte speichereffizient darzustellen. In Java

beispielsweise wird jeder Integer-Wert durch 4 Byte kodiert. Dies bedeutet erstens einen großen Overhead für kleine Zahlen, zweitens schränkt es den Wertebereich auf Zahlen im Intervall von -2.147.483.647 bis 2.147.483.648 ein.

Um diese beiden Nachteile zu umgehen, nutzen wir z.B. die folgende dynamische Überlaufkodierung für ganzzahlige Werte, ebenso können aber auch entsprechende, andere Überlaufkodierungen genutzt werden, die es erlauben, beliebig große Integer-Werte mit dynamischer Bitanzahl darzustellen. Sei n die Anzahl an Bits, die wir für einen Integer-Wert mindestens verwenden wollen, z.B. n=6. Wir nutzen nun für jede Bitfolge von n Bits das allererste Bit als Markierungsbit: Ist das erste Bit ein '1'-Bit, so gehören die nächsten n Bits ebenfalls zu dieser Zahl, ist das erste Bit ein '0'-Bit, so wurde mit den aktuellen n Bits die letzte Bitfolge der Zahl gelesen.

Beispiel 2.1 *Die Binärdarstellung der Zahl 134 ist 10000110. Um eine Überlaufkodierung mit n=6 für diese Zahl zu berechnen, teilen wir sie zunächst in Gruppen der Größe 5 Bits, wir erhalten also: 100 00110. Nach Auffüllen von führenden Nullen erhalten wir: 00100 00110. Um nun auszudrücken, dass beide 5-stelligen Bitfolgen zur selben Zahl gehören, stellen wir der letzten Bitfolge ein '0'-Bit voran, allen übrigen Bitfolgen dieser Zahl stellen wir ein '1'-Bit voran. Die endgültige Kodierung der Zahl 134 mit n=6 ist also: 100100 000110.*

2.7 Beispiel

Im Verlauf dieser Arbeit werde ich alle vorgestellten Verfahren mit Hilfe des in Listing 2.3 vorgestellten Beispiels erläutern. Bei diesem Beispiel-Dokument handelt es sich um eine kleine *Adress*-Datenbank mit derzeit drei *Person*en, die jeweils

- einen *Name*n,
- beliebig viele Fragemente jeweils bestehend aus
  - (*Strasse* und *Ort*) oder *Postfach*,
  - optional einer *Telefon*nummer mit entsprechendem Telefon*modell* (z.B. mobil oder Festnetz)

beinhalten.

```
1 <Adressen>
2     <Person>
3         <Name>Peter Müller</Name>
4         <Postfach>0815</Postfach>
5     </Person>
```

```
6     <Person>
7         <Name>Anna Schmidt</Name>
8         <Strasse>Lindenstrasse</Strasse>
9         <Ort>Berlin</Ort>
10        <Postfach>4711</Postfach>
11    </Person>
12     <Person>
13        <Name>Paul Schulze</Name>
14        <Postfach>3300</Postfach>
15        <Telefon modell="mobil">0171/666666</Telefon>
16    </Person>
17</Adressen>
```

Listing 2.3: XML-Beispiel-Datei

Abbildung 2.1 zeigt die Baum-Darstellung dieses XML-Dokumentes, während Abbildung 2.2 die Binärbaum-Darstellung zeigt.

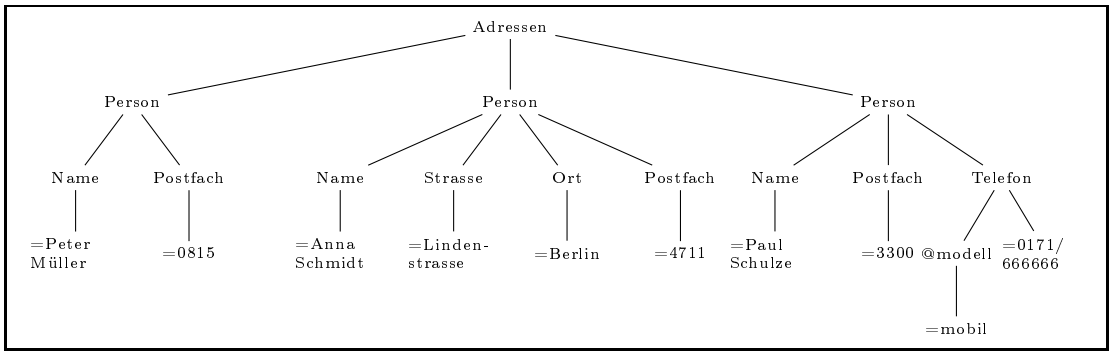

Abbildung 2.1: Baum-Darstellung des XML-Dokumentes

Listing 2.4 enthält die dazugehörige DTD. Diese definiert, dass innerhalb des Elements *Adressen* beliebig viele *Person*-Elemente geschachtelt sind (Zeile 1). Zu einem *Person*-Element sind als Kindknoten

- ein *Name*-Element,
- eine Folge von Fragmenten bestehend aus
  - (*Strasse* und *Ort*) oder *Postfach*,
  - optional einer *Telefon*nummer mit entsprechendem Telefon*modell* (z.B. mobil oder Festnetz)

zugelassen (Zeile 2).

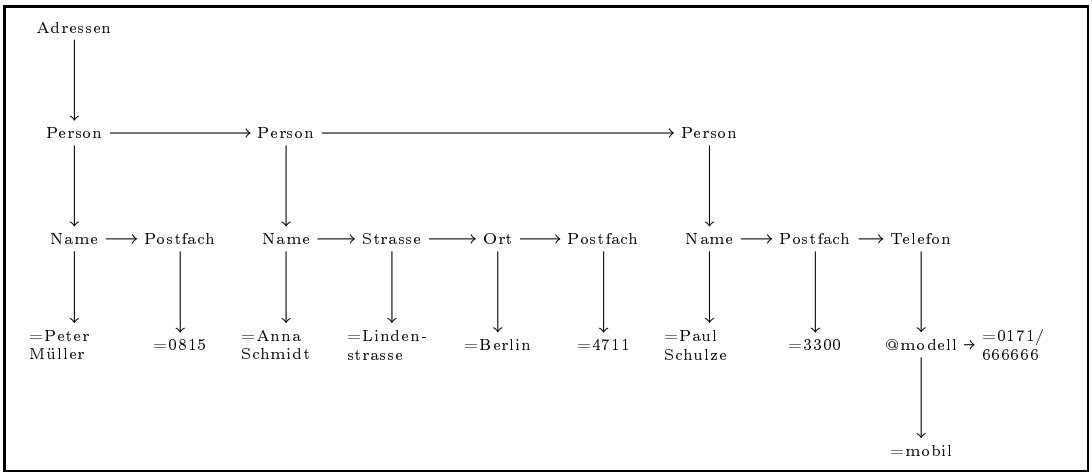

Abbildung 2.2: Binärbaum-Darstellung des XML-Dokumentes

Die Elemente *Strasse*, *Ort*, *Postfach* und *Telefon* enthalten keine Kindelemente, sondern lediglich einen Textknoten als Kind (Zeilen 3-6). *Telefon* hat noch zusätzlich ein Attribut mit Namen *modell* (Zeile 7).

```
1 <!ELEMENT Adressen (Person)*>
2 <!ELEMENT Person (Name, ((Postfach | (Strasse, Ort)),
    Telefon?)*)>
3 <!ELEMENT Strasse #PCDATA>
4 <!ELEMENT Ort #PCDATA>
5 <!ELEMENT Postfach #PCDATA>
6 <!ELEMENT Telefon #PCDATA>
7 <!ATTLIST Telefon modell #REQUIRED>
```

Listing 2.4: DTD zur XML-Beispiel-Datei

3 Ableitbare SAX-Strom-Varianten

3.1 Motivation

Betrachtet man XML-Dokumente, so stellt man fest, dass sich in der Regel im Strukturanteil, also innerhalb der Element- und der Attribut-Knoten, eine größere Anzahl an Wiederholungen findet als im Datenanteil, also innerhalb der Text- und der Attribut-Wert-Knoten. Dies ist darin begründet, dass jedes XML-Dokument nur eine sehr beschränkte Anzahl von verschiedenen Element- und Attributnamen benutzt.

Daher liegt es nahe, zur Kompression von Struktur und Daten jeweils verschiedene Kompressions-Verfahren zu benutzen. In diesem Kapitel stelle ich daher vor, wie man den SAX-Eingabestrom in zwei Eingabeströme aufteilen kann, um diese beiden Ströme dann an getrennte Kompressoren weiterzuleiten. In Kapitel 8 stelle ich dann im Anschluss an die Kompressions-Verfahren für Strukturanteil und Datenanteil vor, wie man die beiden komprimierten Ströme miteinander integrieren kann, um weiterhin eine effiziente Weiterverarbeitung – z.B. in Form von Dekompression oder Anfrage-Auswertung – zu gewährleisten.

3.2 Struktur-Strom

Ein Struktur-Strom enthält nur die Struktur des XML-Dokumentes bzw. des XML-Baumes, also nur den Wurzelknoten sowie die Element- und Attribut-Knoten. Die Text- und Attribut-Wert-Knoten werden in Form eines Platzhalters gespeichert, d.h. als Element-Knoten mit Label '=T'. Die eigentlichen Label der Text-Knoten werden im separaten Daten-Strom gespeichert.

Um die Darstellung zu vereinfachen, werden die Attribut-Knoten als besondere Element-Knoten gespeichert, man kann einen Attribut-Knoten lediglich am Markierungszeichen '@' zu Beginn des Labels erkennen. Auch das startDocument- und das endDocument-Event können als Sonderfälle des start-Element- bzw. endElement-Events gesehen werden, gekennzeichnet durch das

Label 'root', wobei root nicht in der Labelmenge Σ des XML-Baumes enthalten sein darf.

Dies führt uns zu einem einfachen Struktur-Strom, der nur noch die beiden Events startElement und endElement enthält. Formal ist ein Struktur-Strom wie folgt definiert:

Definition 3.1 (Struktur-Strom)**.** Sei *xml* ein XML-Baum nach Definition 2.1 mit Knotenmenge $V = \{r\} \cup VE \cup VA \cup VAW \cup VT$ und Kantenmenge $E \subseteq (V \times V)$ und Labelmenge Σ. Sei SE=$\{$startElement(σ)$|$ $\sigma \in \Sigma\} \cup \{$endElement($\sigma$)$|$ $\sigma \in \Sigma\}$ die Menge aller Struktur-Events. Eine Liste $(s_1, \ldots, s_n)$ mit $s_i \in SE$ für 1≤i≤n bezeichnen wir als *Struktur-Strom*. Zu jedem Knoten $v \in V$ mit Kindknoten $cv_1, \ldots, cv_m$ sei der Struktur-Strom $ss(v) : V \to SE*$ definiert durch

$$ss(v) := \begin{cases} (startElement('root')) \otimes \\ (startElement(v.label)) \otimes ss(cv_1) \otimes \ldots \otimes \\ ss(cv_m) \otimes (endElement(v.label)) \otimes \\ (endElement('root')) & falls\ v = r \\ (startElement(v.label)) \otimes ss(cv_1) \otimes \ldots \otimes \\ ss(cv_m) \otimes (endElement(v.label)) & falls\ v \in VE \\ (startElement('@' + v.label)) \otimes ss(cv_1) \otimes \\ (endElement('@' + v.label)) & falls\ v \in VA \\ (startElement('= T')) \otimes (endElement('= T')) & falls\ v \in (VT \cup VAW) \end{cases}$$

Der Struktur-Strom des XML-Baumes xml ist dann definiert durch $SS(xml) := ss(r)$. □

Im Folgenden bezeichne () die leere Liste.

Definition 3.2 (startElement- und endElement-Strom)**.** Sei *xml* ein XML-Baum nach Definition 2.1 mit Knotenmenge $V = \{r\} \cup VE \cup VA \cup VAW \cup VT$ und Kantenmenge $E \subseteq (V \times V)$ und Labelmenge Σ. Sei SE=$\{$startElement(σ)$|$ $\sigma \in \Sigma\} \cup \{$endElement($\sigma$)$|$ $\sigma \in \Sigma\}$ die Menge aller Struktur-Events. Sei S:=$SS(xml)$=$(s_1, \ldots, s_n)$, mit $s_i \in SE$ für 1≤i≤n der Struktur-Strom von xml nach Definition 3.1.

Dann sei $sstart : \{1, \ldots, n\} \to (SE \cup ())$ definiert durch

$$sstart(x) := \begin{cases} (s_x \in S) & falls\ s_x = startElement(\sigma) \text{ für } \sigma \in \Sigma \\ () & sonst \end{cases}$$

Dann bezeichnen wir $SStart(SS) := (sstart(1) \otimes \ldots \otimes sstart(n))$ als *startElement-Strom von S*.

Sei weiterhin $send : \{1, \ldots, n\} \to (SE \cup ())$ definiert durch

$$send(x) := \begin{cases} (s_x \in S) & falls\ s_x = endElement(\sigma) \text{ für } \sigma \in \Sigma \\ () & sonst \end{cases}$$

Dann bezeichnen wir $SEnd(SS) := (send(1) \otimes \ldots \otimes send(n))$ als *endElement-Strom von S.*

□

Die Berechnung eines Struktur-Stroms zu einem SAX-Event-Strom geht also entsprechend Definitionen 2.2 und 3.1 wie folgt vor:

- *startDocument und endDocument:*
  Die SAX-Events *startDocument* und *endDocument* werden umgewandelt in *startElement('root')* bzw. *endElement('root').*
- *characters*
  Aus einem SAX-Event vom Typ *characters(textwert)* wird eine Event-Folge *startElement('=T')* und *endElement('=T')* generiert. Hierbei steht *'=T'* für einen Platzhalter für Textknoten; *'=T'* darf nicht als Elementname im XML-Dokument benutzt werden.
- *startAttribute und endAttribute*
  Für jedes Paar Attribut=Value wird eine Element-Folge startElement('@' + Attribut), startElement('=T'), endElement('=T'), endElement('@' + Attribut) generiert.
- *startElement und endElement*
  Die SAX-Events startElement und endElement werden unverändert in den Struktur-Strom geschrieben.

Der Platzhalter *'=T'* gewährleistet hierbei, dass man bei der späteren Verarbeitung rekonstruieren kann, an welcher Stelle im Dokument ein Textknoten enthalten war. Die Reihenfolge der Textknoten ist hierbei durch die Reihenfolge der Werte innerhalb des Daten-Stroms gegeben.

Beispiel 3.1 *Listing 3.1 zeigt den Struktur-Strom der zum Beispieldokument aus Listing 2.3 generiert wird.*

```
1 startElement('root');
2 startElement('Adressen');
3 startElement('Person');
4 startElement('Name');
5 startElement('=T');
6 endElement   ('=T');
7 endElement   ('Name');
```

```
 8 startElement('Postfach');
 9 startElement('=T');
10 endElement   ('=T');
11 endElement   ('Postfach');
12 endElement   ('Person');
13 startElement('Person');
14 startElement('Name');
15 startElement('=T');
16 endElement   ('=T');
17 endElement   ('Name');
18 startElement('Strasse');
19 startElement('=T');
20 endElement   ('=T');
21 endElement   ('Strasse');
22 startElement('Ort');
23 startElement('=T');
24 endElement   ('=T');
25 endElement   ('Ort');
26 startElement('Postfach');
27 startElement('=T');
28 endElement   ('=T');
29 endElement   ('Postfach');
30 endElement   ('Person');
31 startElement('Person');
32 startElement('Name');
33 startElement('=T');
34 endElement   ('=T');
35 endElement   ('Name');
36 startElement('Postfach');
37 startElement('=T');
38 endElement   ('=T');
39 endElement   ('Postfach');
40 startElement('Telefon');
41 startElement('@modell');
42 startElement('=T');
43 endElement   ('=T');
44 endElement   ('@modell');
45 startElement('=T');
46 endElement   ('=T');
47 endElement   ('Telefon');
48 endElement   ('Person');
```

```
49 endElement   ('Adressen');
50 endElement   ('root');
```

Listing 3.1: Simple-SAX-Strom des Beispiels

3.2.1 Unterscheidung von Elementen, Attributen und Text-Werten bei der Anfrage-Auswertung

Im Struktur-Strom werden Elemente, Attribute und Text-Werte gleich behandelt und als Element mit gegebenfalls besonderer Markierung betrachtet. Bei der Anfrage-Auswertung werden jedoch die drei Knoten-Typen über verschiedene Achsen angesprochen.

Daher muss bei der Anfrage-Auswertung die Unterscheidung zwischen Elementen, Attributen und Text-Werten sichergestellt werden. Lautet die Anfrage z.B. /*, es sind also alle Element-Kindknoten gesucht, nicht aber die Attribute, so müssen die Attribute, also diejenigen Knoten, deren Label mit '@' beginnt, übersprungen werden, und nur die tatsächlichen Element-Knoten als Ergebnis der Anfrage betrachtet werden. Entsprechend müssen bei einer Anfrage /@*, welche alle Attribut-Knoten des aktuellen Kontext-Knotens abfragt, alle Elemente übersprungen werden, und nur die Attribut-Knoten dürfen als Ergebnis betrachtet werden.

3.3 Binärer Struktur-Strom

Analog wie man zu einem XML-Baum die Binärbaum-Darstellung berechnen kann, kann man zu einem Struktur-Strom den binären Struktur-Strom berechnen.

Hierzu werden aus den Events startElement und endElement des Struktur-Stroms die binären Events firstChild, nextSibling und parent des binären Struktur-Stroms generiert. Es werden immer Paare von Struktur-Strom-Events betrachtet.

- *startElement, startElement*
  Ein SAX-Event startElement(x) gefolgt von einem weiteren SAX-Event startElement(a) entspricht der first-child-Achse. Daher wird dieses transformiert in das binäre SAX-Event firstChild(a).
- *startElement(a), endElement(a)*
  Ein SAX-Event startElement(a) gefolgt von einem SAX-Event endElement(a) entspricht einem leeren Element-Tag. Hieraus wird kein binäres SAX-Event generiert.

- *endElement(x), startElement(a)*
  Ein SAX-Event endElement(x) gefolgt von einem SAX-Event startElement(a) entspricht der next-sibling-Achse. Daher wird dieses transformiert in das binäre SAX-Event nextSibling(a).
- *endElement(x), endElement(y)*
  Ein SAX-Event endElement(x) gefolgt von einem weiteren SAX-Event endElement(y) entspricht der parent-Achse. Daher wird dieses transformiert in das binäre SAX-Event parent().

Wir erhalten so einen binären Struktur-Strom, der aus Events vom Typ firstChild, nextSibling und parent besteht. Dieser ist wie folgt formal definiert:

Definition 3.3 (binärer Struktur-Strom)**.** Sei xml ein XML-Baum nach Definition 2.1 mit Labelmenge Σ und $SS(xml) = (ss_1, \ldots, ss_n)$ der Struktur-Strom zu xml nach Definition 3.1. Seien SStart und SEnd startElement- und endElement-Strom von SS(xml) nach Definition 3.2. Sei weiterhin BE=$\{\text{firstChild}(\sigma) |\ \sigma \in \Sigma\} \cup \{\text{nextSibling}(\sigma) |\ \sigma \in \Sigma\} \cup \{\text{parent}()\}$ die Menge aller binären Struktur-Events. Dann ist die Abbildung
$bs_{SS(xml)} : \{1, \ldots, n-1\} \rightarrow (BE \cup ())$ definiert durch:

$$bs_{SS(xml)}(x) := \begin{cases} (firstChild(\sigma)) & \text{falls } ss_{x+1} = startElement(\sigma) \wedge ss_x \in SStart \\ (nextSibling(\sigma)) & \text{falls } ss_{x+1} = startElement(\sigma) \wedge ss_x \in SEnd \\ (parent()) & \text{falls } ss_{x+1} \in SEnd \wedge ss_x \in SEnd \\ () & sonst \end{cases}$$

Der binäre Struktur-Strom des XML-Baumes xml bzw. des Struktur-Stromes $SS(xml) = (ss_1, \ldots, ss_n)$ ist dann definiert durch
$BS := (bs(1) \otimes \ldots \otimes bs(n-1))$. □

Beispiel 3.2 *Listing 3.2 zeigt den binären SAX-Strom, der zum Beispieldokument aus Listing 2.3 bzw. zum Struktur-Strom aus Listing 3.1 generiert wird.*

```
1 firstChild('Adressen');
2 firstChild('Person');
3 firstChild('Name');
4 firstChild('=T');
5 parent();
6 nextSibling('Postfach');
```

```
 7 firstChild('=T');
 8 parent();
 9 parent();
10 nextSibling('Person');
11 firstChild('Name');
12 firstChild('=T');
13 parent();
14 nextSibling('Strasse');
15 firstChild('=T');
16 parent();
17 nextSibling('Ort');
18 firstChild('=T');
19 nextSibling('Postfach');
20 firstChild('=T');
21 parent();
22 parent();
23 nextSibling('Person');
24 firstChild('Name');
25 firstChild('=T');
26 parent();
27 nextSibling('Postfach');
28 firstChild('=T');
29 parent();
30 nextSibling('Telefon');
31 firstChild('@modell');
32 firstChild('=T');
33 parent();
34 nextSibling('=T');
35 parent();
36 parent();
37 parent();
38 parent();
```

Listing 3.2: binärer SAX-Strom des Beispiels

3.4 Daten-Strom

Der Daten-Strom enthält eine Sequenz aller Text-Werte – also Text- und Attribut-Wert-Knoten – in Dokumentreihenfolge.

Da jedoch das umgebende Element bzw. Attribut gleichzeitig eine semantische Kontextinformation beinhaltet, die später zur Optimierung der Kompres-

sion des Daten-Stroms verwendet werden kann, wird dieser Daten-Strom um diesen Element- bzw. Attributnamen angereichert, damit diese Information bei der späteren Kompression zur Verfügung steht. Zur lediglichen Rekonstruktion des Ursprungs-SAX-Stroms sind die Informationen über den Element- bzw. Attributnamen nicht notwendig. Der Daten-Strom enthält also Paare (Name, Text-Wert) aus Element- bzw. Attributnamen und Text-Werten.

Formal kann man den Daten-Strom zu einem XML-Dokument bzw. XML-Baum wie folgt definieren. Hierbei bezeichnet der Operator $\otimes$ die Konkatenation zweier Listen.

Definition 3.4 (Daten-Strom)**.** Sei *xml* ein XML-Baum nach Definition 2.1 mit Knotenmenge $V = \{r\} \cup VE \cup VA \cup VAW \cup VT$ und Kantenmenge $E \subseteq (V \times V)$ und Labelmenge Σ.

Zu einem Knoten $v \in V$ mit Kindknoten $cv_1, \ldots, cv_n$ und Elternknoten pv sei der Datenstrom $data(v)$ definiert durch

$$data(v) := \begin{cases} data(cv_1) \otimes \ldots \otimes data(cv_n) & falls\ v \in \{r\} \cup VE \cup VA \\ (pv.label, v.label) & falls\ v \in VT \cup VAW \end{cases}$$

Der Daten-Strom des XML-Baumes xml ist dann definiert durch $DATA(xml) := data(r)$. □

Beispiel 3.3 *Listing 3.3 zeigt den entsprechenden Daten-Strom, der zum Beispieldokument aus Listing 2.3 generiert wird.*

```
1  ('Name',       'Peter Müller');
2  ('Postfach',   '0815');
3  ('Name',       'Anna Schmidt');
4  ('Strasse',    'Lindenstrasse');
5  ('Ort',        'Berlin');
6  ('Postfach',   '4711');
7  ('Name',       'Paul Schulze');
8  ('Postfach',   '3300');
9  ('@modell',    'mobil');
10 ('Telefon',    '0171/666666');
```

Listing 3.3: Daten-Strom des Beispiels

4 XML-Kompression durch platzeffiziente Kodierung

Das erste vorgestellte Verfahren – das *Succinct-Verfahren* – bietet eine komprimierte Darstellung der XML-Struktur, indem es die Struktur durch kürzere Zeichenfolgen darstellt. Im ersten Abschnitt dieses Kapitels werde ich zunächst das zugrunde liegende Verfahren aus [48] vorstellen und die diesem Verfahren zugrunde liegenden Ideen formalisieren. Anschließend werde ich dann eine Optimierung der Darstellung vorstellen, die es erlaubt, die Anfrageauswertung effizienter zu gestalten. Schließlich werde ich Update-Operationen auf dem optimierten Verfahren erläutern.

4.1 Succinct-Darstellung und die atomaren XPath-Achsen

4.1.1 Succinct-Darstellung

Das Succinct-Verfahren erhält als Eingabe den Struktur-Strom nach Definition 3.1 und berechnet daraus

- einen *Bitstrom*, der die Schachtelung der Start- und End-Tags, nicht aber deren Label darstellt,
- eine *Symboltabelle*, die eine Zuordnung von Labeln zu Symbolen – also kurzen Bit-Darstellungen – enthält,
- einen *Symbol-Strom*, der eine Zuordnung von Bitstrom-Positionen zu Symbolen enthält.

Die folgenden Definitionen 4.1 bis 4.7 beschreiben die Datenstrukturen, Hilfsfunktionen und Berechnungsfunktionen zur Kompression, und die anschließenden Definitionen 4.8 bis 4.10 beschreiben Berechnungsfunktionen zur Dekompression.

Definition 4.1 (Bitstrom, Position)**.** Sei Σ die Menge aller Label. Sei SE=$\{$startElement(σ)$|\ \sigma \in \Sigma\} \cup \{$endElement($\sigma$)$|\ \sigma \in \Sigma\}$ die Menge aller Struktur-Events. Sei S=$(s_1, \ldots, s_n)$ mit $s_i \in SE$ für 1≤i≤n ein Struktur-Strom nach Definition 3.1 und SStart der startElement-Strom von S nach Definition 3.2. Sei die Funktion b: SE $\rightarrow \{0,1\}$ definiert als

$$\text{b(s)} := \begin{cases} 1 & falls\ s \in SStart \\ 0 & sonst \end{cases}$$

Dann bezeichnen wir die Bitfolge B(S)=(b(s_1), ..., b(s_n)) als *Bitstrom* zum Struktur-Strom S. Die Indizes 1, ..., n nennen wir *Positionen* der Events $s_1, \ldots, s_n$ im Bitstrom. Weiterhin bezeichnen wir mit e(B(S)):=$\{$i $\mid b(s_i)= 1\}$ die Menge aller *Eins-Positionen* von B(S) und mit n(B(S)):=$\{$i $\mid b(s_i)=0\}$ die Menge aller *Null-Positionen* von B(S).

Beispiel 4.1 *Listing 4.1 zeigt den Bitstrom, der zum Struktur-Strom aus Listing 3.1 generiert wird.*

```
1 1 1 1 1 0 0 1 1 0 0 0 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
    0 0 1 1 1 0 0 1 1 0 0 1 1 1 1 0 0 1 0 0 0 0 0 0
```

Listing 4.1: Bitstrom des Beispiels

Analog zu einem wohlgeformten XML-Dokument werde ich einen *korrekten Bitstrom* definieren. Hierzu werden zunächst einige Hilfsfunktionen benötigt, die gewisse Bitstrom-Eigenschaften repräsentieren.

Die Funktion *level* berechnet die Tiefe eines durch ein '1'-Bit im Bitstrom repräsentierten XML-Knotens.

Definition 4.2 (Level)**.** Sei S ein Struktur-Strom und B=$(b_1, \ldots, b_n)$ der Bitstrom zu S nach Definition 4.1.

Sei die Hilfsfunktion v_B: $\{1, \ldots, \text{n}\} \rightarrow \{\text{-}1,1\}$ definiert durch

$$v_B\text{(x)} := \begin{cases} 1 & falls\ x \in e(B) \\ -1 & sonst \end{cases}$$

Dann ist die Funktion $level_B$: $\{1, \ldots, \text{n}\} \rightarrow \{0, \ldots, \text{n}\}$ definiert durch
$level_B\text{(x)} := \Sigma_{i=1}^{x} v_B(x)$

Die Funktion end berechnet die Position des zugehörigen endElement-Events zu einer gegebenen Position eines gegebenen startElement-Events. Die Funktion start entspricht der Umkehrung der Funktion end und berechnet die Position des zugehörigen startElement-Events zu einer gegebenen Position eines gegebenen endElement-Events.

Definition 4.3 (End, Start)**.** Sei B=$(b_1, \ldots, b_n)$ ein Bitstrom nach Definition 4.1 mit der Menge e(B) der Eins-Positionen und der Menge n(B) der Null-Positionen. Dann ist die Funktion end_B: e(B) $\rightarrow$ n(B) definiert durch

end_B(x) :=y mit y>x $\wedge$ $level_B$(x)-$level_B$(y)=1 $\wedge$ *not* $\exists$ z: $level_B$(x)-$level_B$(z)=1 $\wedge$ x<z<y.

Entsprechend ist die Funktion $start_B$: n(B) $\rightarrow$ e(B) definiert durch

$start_B$(x):= y mit y<x $\wedge$ $level_B$(y)-$level_B$(x)=1 $\wedge$ *not* $\exists$ z: $level_B$(y)-$level_B$(z)=1 $\wedge$ y<z<x.

Satz 4.1. Sei S ein Struktur-Strom, sei E ein Element, und sei i die Position des startElement-Events von E in S. Dann gilt: j ist die Position des endElement-Events von E in S $\Leftrightarrow$ end_B(i)=j.

Beweis. Folgt aus der Analogie von Bitstrom und Struktur-Strom nach den Definitionen 3.1, 4.1, 4.2 und 4.3. □

Die Funktion endOfParent berechnet die Position des endElement-Events des parents zu einer gegebenen Position eines startElement-Events.

Definition 4.4 (EndOfParent)**.** Sei B=(b_1, ..., b_n) ein Bitstrom nach Definition 4.1 mit der Menge e(B) der Eins-Positionen und der Menge n(B) der Null-Positionen. Dann ist die Funktion $endOfParent_B$: e(B) $\rightarrow$ n(B) definiert durch

$endOfParent_B$(x) :=y mit y>x $\wedge$ $level_B$(x)-$level_B$(y)=2 $\wedge$ *not* $\exists$ z: $level_B$(x)-$level_B$(z)=2 $\wedge$ x<z<y

Satz 4.2. Sei S ein Struktur-Strom nach Definition 3.1, sei E ein Element, sei i die Position des startElement-Events von E in S, und sei P=E/parent der parent-Knoten von E. Dann gilt: j ist die Position des endElement-Events von P in S $\Leftrightarrow$ endOfParent(i)=j.

Beweis. Folgt aus Definitionen 2.5, 4.1, 4.2 und 4.4. □

Anmerkung: Die Funktion $endOfParentB$ kann auch als Konkatenation der Funktionen end_B und der später vorgestellten Funktion parent definiert werden. Da jedoch die Funktion parent eine „Rückwärtsnavigation“ im Bitstrom darstellt, wurde hier eine andere Definition gewählt, die eine lineare Durchquerung des Bitstroms erlaubt.

Definition 4.5 (Korrekter Bitstrom)**.** Sei B=(b_1, ..., b_n) ein Bitstrom nach Definition 4.1 mit den Funktionen $level_B$, end_B und $endOfParent_B$. Dann bezeichnen wir B als *korrekten Bitstrom* genau dann, wenn

1. $\forall$ x $\in$ e(B) $\exists$ y $\in$ n(B): end_B(x)=y,
2. $\forall$ y $\in$ {1, ..., n-1}: $level_B$(y)>0.

Definition 4.6 (Symboltabelle)**.** Sei Σ eine Menge von Labeln und Sym eine Menge von Symbolen mit |Sym|=|Σ|. Eine *Symboltabelle* S ist eine umkehrbare Funktion S: $\Sigma \rightarrow$ Sym, die jedem Label *s* eindeutig ein Symbol *S(s)* zuordnet.

Beispiel 4.2 *Tabelle 4.1 zeigt eine mögliche Symboltabelle zum Struktur-Strom aus Listing 3.1.*

String	Symbol
root	1
Adressen	2
Person	3
Name	4
=T	5
Postfach	6
Strasse	7
Ort	8
Telefon	9
@modell	10

Tabelle 4.1: Symboltabelle des Beispiels

Definition 4.7 (Symbol-Strom)**.** Sei Σ die Menge aller Label, Sym eine Menge von Symbolen mit $|\text{Sym}|=|\Sigma|$, ST:$\Sigma \rightarrow$ Sym eine Symboltabelle. Sei start= $\{\text{startElement}(\sigma) \mid \sigma \in \Sigma\}$ die Menge aller startElement-Events über Σ. Dann ist die umkehrbare Funktion *ss*: start $\rightarrow$ Sym definiert als

ss(startElement(σ)) :=ST(σ).

Sei S=(s_1, ..., s_n) ein Struktur-Strom nach Definition 3.1 und sei SStart= ($sstart_1$, ..., $sstart_{\frac{n}{2}}$) der startElement-Strom von S nach Definition 3.2. Die Liste von Symbolen SS(S)=(ss($sstart_1$), ..., ss($sstart_{\frac{n}{2}}$)) bezeichnen wir als *Symbol-Strom* zum Struktur-Strom S.

Beispiel 4.3 *Listing 4.2 zeigt den Symbol-Strom zum Struktur-Strom aus Listing 3.1 entsprechend der Symboltabelle aus Tabelle 4.1.*

```
1 2 3 4 5 6 5 3 4 5 7 5 8 5 6 5 3 4 5 6 5 3 4 5 6 5 9
  10 5 5
```

Listing 4.2: Symbol-Strom des Beispiels

Zusammen stellen Bitstrom, Symboltabelle und Symbol-Strom eine speichereffiziente Darstellung der Struktur eines XML-Dokuments dar.

Während die Dekompression des Bitstroms zu einer Folge von start- und endElement-Events – ohne Label-Information – direkt durch die Umkehrung der Funktion b gelöst werden kann, erfordert die Ermittlung der Label zu diesen Events einen höheren Berechnungsaufwand.

Zur Berechnung der Label-Information wird die Funktion rang benötigt, die zu einer gegebenen Position im Bitstrom berechnet, wie viele '1'-Bits im Bitstrom bis zu dieser Position vorhanden sind.

Definition 4.8 (Rang)**.** Sei B=(b_1, ..., b_n) ein Bitstrom. Dann ist die Funktion $rang_B$: {1, ..., n} → {1, ..., n} definiert als: $rang_B$(x) :=$\Sigma_{i=1}^{x} b_i$

Die Funktion *label* berechnet das Label zu einem durch eine '1'-Position im Bitstrom gegebenen startElement-Event.

Definition 4.9 (Label)**.** Sei Σ eine Menge von Labeln, Sym eine Menge von Symbolen mit |Sym|=|Σ|, ST:Σ → Sym eine Symboltabelle. Sei B=(b_1, ..., b_n) ein korrekter Bitstrom, und sei SS=(ss_1, ..., $ss_{\frac{n}{2}}$) ein Symbol-Strom. Dann ist die Funktion $label_{SS}$: e(B)→ Σ definiert als

$label_{SS}$(x):=$ST^{-1}(ss_{rang_B(x)})$.

Wir sagen auch $label_{SS}$(x) ist *das Label zur 1-Position x bzw. zum zur 1-Position gehörenden startElement-Events.*

Satz 4.3. Sei S ein Struktur-Strom, sei i die Position eines startElement-Events sE in S und lab ein Label. Dann gilt: lab ist das Label von sE ⇔ $label_{SS}$(i)=lab.

Beweis. Folgt aus Definitionen 4.7, 4.8 und 4.9. □

Mit Hilfe dieser Funktionen wird die Dekompressions-Funktion wie folgt definiert:

Definition 4.10 (Decomp)**.** Sei Σ eine Menge von Labeln, Sym eine Menge von Symbolen mit |Sym|=|Σ|, ST:Σ → Sym eine Symboltabelle. Sei B=(b_1, ..., b_n) ein korrekter Bitstrom, und sei SS=(ss_1, ..., $ss_{\frac{n}{2}}$) ein Symbol-Strom. Sei weiterhin E:={startElement(σ)|$\sigma \in \Sigma$} ∪ {endElement(σ)|$\sigma \in \Sigma$} die Menge aller Events über Σ.

Dann ist die Funktion $decomp_{SS,B}$: {1, ..., n} → E definiert als

$$decomp_{SS,B}(\text{x}) := \begin{cases} startElement(label_{SS}(x)) & falls\ x \in e(B) \\ endElement(label_{SS}(start_B(x))) & sonst \end{cases}$$

Der folgende Satz besagt, dass Bitstrom, Symboltabelle und Symbol-Strom zusammen mit der Funktion decomp eine korrekte Kompression darstellen. Dies bedeutet, dass wenn man zu einem Struktur-Strom S und einer Symboltabelle ST den Bitstrom B und den Symbol-Strom SS berechnet und anschließend die Funktion decomp auf B und SS ausführt, dann erhält man wieder den ursprünglichen Struktur-Strom S.

Satz 4.4. Sei Σ eine Menge von Labeln, Sym eine Menge von Symbolen mit |Sym|=|Σ|, ST:Σ → Sym eine Symboltabelle. Sei S=(s_1, ..., s_n) ein Struktur-Strom und SStart der startElement-Strom von S. Sei weiterhin

B(S)=(b_1, ..., b_n) der Bitstrom zu S und SS(SStart)=(ss_1, ..., $ss_{\frac{n}{2}}$) der Symbol-Strom zu S und ST. Dann gilt $decomp_{SS,B}$(i)=s_i für alle 1≤i≤n.

Beweis. Sei s_i vom Typ startElement(lab). Dann ist b_i=1 also i ∈ e(B) und $ss_{rang_B(i)}$=ST(lab). Somit gilt $label_{SS}$(i)=$ST^{-1}(ss_{rang_B(i)})$=lab und $decomp_{SS,B}$(i)=startElement(lab).

Sei nun s_i vom Typ endElement(lab). Dann ist b_i=0, also i ∉ e(B). Das Label lab steht in der Labelliste an der Position des dazugehörigen startElement-Events, also $ss_{rang_B(start_B(i))}$=ST(lab). Somit gilt $label_{SS}(start_B$(x))= $ST^{-1}(ss_{rang_B(start_B(i))})$=lab und $decomp_{SS,B}$(i)=endElement(lab).

□

Durch die Verwendung der Funktion start erfordert die Dekompression eine Rückwärtsnavigation, die in vielen Fällen unerwünscht ist. Dies kann vermieden werden, indem für alle geöffneten, aber noch nicht geschlossenen Knoten Kx an Position x der Wert der Funktion label(start(x)) auf einem Stack zwischengespeichert wird. Da die geöffneten, aber noch nicht geschlossenen Knoten dem child-Pfad von der Wurzel bis zum aktuellen Knoten entsprechen, ist die Menge der zu speichernden Werte beschränkt durch die Dokument-Tiefe. Dennoch gilt, dass zwar die Kompression ohne weiteren Speicherbedarf durch lineares Durchqueren durchgeführt werden kann, zur Dekompression wird aber entweder bidirektionales Durchqueren oder weiterer Speicherbedarf der Größe O(n) benötigt, wobei n die maximale Tiefe des Dokuments ist.

4.1.2 Abbilden der atomaren Achsen first-child und next-sibling

In diesem Kapitel werde ich nun erläutern, wie man mit Hilfe des Bitstroms der Succinct-Darstellung entlang der Achsen first-child und next-sibling navigieren kann, ohne vorher das ursprüngliche XML-Dokument zu rekonstruieren.

Hat ein Knoten p ein first-child fc, so befindet sich das startElement-Event von fc im Struktur-Strom direkt hinter dem startElement-Event von p.

Dies führt zur folgenden Definition der Funktion fc, die zu einer gegebenen Position die eventuelle Position des first-childs berechnet.

Definition 4.11 (Funktion fc)**.** Sei B=(b_1, ..., b_n) ein korrekter Bitstrom mit der Menge e(B) der Eins-Positionen. Dann ist die Funktion *fc*: e(B) → {1, ..., n} definiert als:
fc(x) :=x+1

Satz 4.5. Sei S ein Struktur-Strom und seien i und j die Positionen je eines startElement-Events eines Knotens k_i bzw. k_j im Struktur-Strom und B(S)=(b_1, ..., b_n) der Bitstrom von S. Dann gilt: k_j=first-child(k_i) ⇔ fc(i)=j und b_j=1.

Beweis. Beweis folgt aus Definitionen 2.4(a), 4.1 und 4.11. □

Das next-sibling eines Knotens k ist der Knoten, der auf den End-Tag von k folgt. Dies führt zu der folgenden Funktion ns, die zu einer gegebenen Position aus der Menge der Eins-Positionen die eventuelle Position des next-siblings berechnet.

Definition 4.12 (Funktion ns)**.** Sei B=(b_1, ..., b_n) ein korrekter Bitstrom mit der Menge e(B) der Eins-Positionen. Dann ist die Funktion *ns*: e(B) → {1, ..., n+1} definiert als:
ns(x) :=end_B(x) + 1.

Satz 4.6. Sei S ein Struktur-Strom und seien i und j die Positionen je eines startElement-Events eines Knotens k_i bzw. k_j im Struktur-Strom und B(S)=($b_1, \ldots, b_n$) der Bitstrom von S. Dann gilt: k_j=next-sibling(k_i) ⇔ ns(i)=j und b_j=1.

Beweis. Beweis folgt aus Definitionen 2.4(b), 4.1 und 4.12. □

Satz 4.7. Sei B=(b_1, ..., b_n) ein korrekter Bitstrom mit der Menge e(B) der Eins-Positionen. Dann kann
(a) die Funktion fc in O(1) und
(b) die Funktion ns in O(n) berechnet werden.

Beweis. (a) folgt aus Definition 4.11.
Da die Berechnung der Position end_B im worst case ein komplettes Durchqueren des Bitstroms erfordert, folgt Aussage (b) aus Definition 4.12. □

Laut den Sätzen 4.3, 4.5, 4.6 und 4.7 ermöglichen die Funktionen label, ns und fc eine effiziente Anfrage-Auswertung direkt auf der Succinct-Darstellung ohne vorherige Dekompression. Dabei können die Anfragen so ausgewertet werden, dass ein einmaliges lineares Durchqueren des Bitstroms und der Labelliste genügt; diese Anfrage-Auswertung ist auch für quasi-unendliche, komprimierte Datenströme geeignet.

4.2 Optimierte Auswertung der Vorwärts-Achsen

Nachdem ich im vorangehenden Kapitel die Ideen aus [48] vorgestellt und formalisiert habe, werde ich nun eine Optimierung dieser Ideen präsentieren, die eine optimierte Auswertung der Vorwärtsachsen child, descendant, following-sibling und following erlaubt.

Hauptidee dieser Optimierung ist die Benutzung einer *invertierten* Labelliste an Stelle der Labelliste, also einer Zuordnung von '1'-Bit-Positionen zu Labeln.

Definition 4.13 (Invertierte Labellisten)**.** Sei Σ die Menge aller Label. Sei $\sigma \in \Sigma$ ein beliebiges Label. Sei S=(s_1, ..., s_n) ein Struktur-Strom, sei SStart der startElement-Strom von S und sei $start_\sigma$:=($s_i \in$ SStart | s_i=startElement(σ)) eine Teilfolge von SStart mit $start_\sigma$=(s_1, ..., s_m). Dann ist die Funktion $invLL_\sigma$: $start_\sigma \rightarrow$ {1, ..., n} definiert durch

$invLL_\sigma$(x):=(i | x=s_i)

Die Liste IL_σ:=($invLL_\sigma(s_1)$, ..., $invLL_\sigma(s_m)$) zu einem Label $\sigma \in \Sigma$ bezeichnen wir als *invertierte Labelliste zu* σ. Die Menge IL_Σ= $\{IL_\sigma \mid \sigma \in \Sigma\}$ bezeichnen wir als die *invertierten Labellisten zu* Σ *und S.*

Beispiel 4.4 *Tabelle 4.2 zeigt die invertierten Labellisten, die aus dem Struktur-Strom aus Listing 3.1 generiert werden.*

$IL_{@modell}$	(41)
$IL_{Adressen}$	(2)
IL_{Name}	(4, 14, 32)
IL_{Ort}	(22)
IL_{Person}	(3, 13, 31)
$IL_{Postfach}$	(26, 36)
$IL_{Strasse}$	(18)
$IL_{Telefon}$	(40)

Tabelle 4.2: Invertierte Labellisten des Beispiels

Durch die Funktion invToSS kann zu einer Menge von invertierten Labellisten ein Symbol-Strom berechnet werden, der dann zur Dekompression genutzt werden kann.

Definition 4.14 (Funktion invToSS)**.** Sei Σ die Menge aller Label, sei Sym eine Menge von Symbolen mit |Sym|=|Σ|, sei ST:$\Sigma \rightarrow$ Sym eine Symboltabelle, sei S ein Struktur-Strom, und sei B(S)=(b_1, ..., b_n) ein Bitstrom. Seien IL_Σ die Menge der invertierten Labellisten zu Σ und S. Dann ist die Funktion $invToSS_{IL_\Sigma}$: $\{1, \ldots, \frac{n}{2}\} \rightarrow$ Sym definiert als

$invToSS_{IL_\Sigma}$(x)=(ST(σ) | x=$rang_B$(y) $\wedge$ y $\in IL_\sigma$).

Laut dem folgenden Satz 4.8 stellt die Kombination aus Bitstrom und invertierten Labellisten eine korrekte Kompression des Struktur-Stroms dar, die mit Hilfe der Funktionen invToSS und decomp umgekehrt werden kann.

Satz 4.8. Sei Σ die Menge aller Label, Sym eine Menge von Symbolen mit |Sym|=|Σ| und ST:$\Sigma \rightarrow$ Sym eine Symboltabelle. Sei S=(S_1, ..., S_n) ein Struktur-Strom über Σ und B(S) der Bitstrom von S. Sei weiterhin IL_Σ die Menge der invertierten Labellisten zu Σ und S. Dann gilt:

SS:=($invToSS_{IL_\Sigma}(1), \ldots, invToSS_{IL_\Sigma}(\frac{n}{2})$)=($ss_1, \ldots, ss_{\frac{n}{2}}$) ist der Symbol-Strom zu ST und S, so dass $decomp_{SS,B}$(i)=s_i für alle 1≤i≤n.

Beweis. Sei s_i vom Typ startElement(lab). Dann ist b_i=1 also i ∈ e(B(S)) und i ∈ IL_{lab}. Somit gilt laut Definition 4.14 $ss_{rang_B(i)}$=$invToSS_{IL_\Sigma}(rang_B$(i))= ST(lab). Weiterhin gilt $label_{SS}$(i)=$ST^{-1}(ss_{rang_B(i)})$=lab und somit $decomp_{SS,B}$(i)=startElement(lab).

Sei nun s_i vom Typ endElement(lab). Dann folgt die Richtigkeit analog zum Beweis zu Satz 4.4. □

Die invertierten Labellisten stellen einen Index auf die '1'-Bit-Positionen sortiert nach den zugehörigen Labeln dar. Dieser Index wird nun bei der optimierten Anfrageauswertung benutzt, um entsprechend des Knoten-Tests eines Location-Steps vorab eine Menge von potentiellen Kandidaten zu ermitteln, so dass die Achsen-Tests nur für diese Kandidaten, nicht aber für alle '1'-Bit-Positionen durchgeführt werden müssen.

4.2.1 child::a

Sei Kp der aktuelle Kontextknoten und p die Position des '1'-Bits, das Kp im Bitstrom B repräsentiert. Sei weiterhin Kc ∈ Kp/child ein child-Knoten von Kp und c die Position des '1'-Bits, das Kc im Bitstrom repräsentiert. Dann muss Kc den Knoten-Test 'a' erfüllen, eine Ebene unterhalb von Kp liegen, und c muss zwischen den Positionen p und end_B(p) liegen, also p<c<end_B(p).

Definition 4.15 ($child_\sigma(p)$)**.** Sei Σ die Menge aller Label und sei σ ∈ Σ ein beliebiges Label. Sei B=($b_1, \ldots, b_n$) ein Bitstrom, sei p eine Position in B und sei IL_σ die invertierte Labelliste zu σ. Dann ist die Menge $child_\sigma(p) \subseteq IL_\sigma$ definiert als:

$child_\sigma$(p) :={c ∈ IL_σ | $level_B$(p)-$level_B$(c)=1 ∧ p<c<end_B(p)}

Satz 4.9. Seien S ein Struktur-Strom, i und j die Positionen je eines startElement-Events eines Knotens k_i bzw. k_j im Struktur-Strom und B(S)=($b_1, \ldots, b_n$) der Bitstrom von S. Dann gilt:

k_j ∈ k_i/child::a ⇔ j ∈ $child_a$(i).

Beweis. Beweis folgt aus Definitionen 2.5(a) und 4.15. □

4.2.2 descendant::a

Sei Kanc der aktuelle Kontextknoten und anc die Position des '1'-Bits, das Kanc im Bitstrom B repräsentiert. Sei weiterhin Kd ∈ Kanc/descendant ein descendant-Knoten von Kanc und d die Position des '1'-Bits, das Kd im Bitstrom repräsentiert. Dann muss Kd den Knoten-Test 'a' erfüllen, und d muss

zwischen den Positionen anc und end_B(anc) liegen, also anc<d<end_B(anc). Es existiert keine Bedingung an $level_B$(d).

Definition 4.16 ($descendant_\sigma(a)$)**.** Sei Σ die Menge aller Label und sei $\sigma \in \Sigma$ ein beliebiges Label. Sei B=(b_1, ..., b_n) ein Bitstrom, sei a eine Position im Bitstrom und sei IL_σ die invertierte Labelliste zu σ. Dann ist die Menge $descendant_\sigma(a) \subseteq IL_\sigma$ definiert als:
$descendant_\sigma$(a) :={d $\in IL_\sigma$ | a<d<end_B(a)}

Satz 4.10. Seien S ein Struktur-Strom, i und j die Positionen je eines startElement-Events eines Knotens k_i bzw. k_j im Struktur-Strom und B(S)=(b_1, ..., b_n) der Bitstrom von S. Dann gilt:
$k_j \in k_i$/descendant::a $\Leftrightarrow$ j $\in descendant_a$(i).

Beweis. Beweis folgt aus Definitionen 2.5(c) und 4.16. □

4.2.3 following-sibling::a

Sei Kp der aktuelle Kontextknoten und p die Position des '1'-Bits, das Kp im Bitstrom B repräsentiert. Sei weiterhin Kf $\in$ Kp/following-sibling ein following-sibling-Knoten von Kp und f die Position des '1'-Bit, das Kf im Bitstrom repräsentiert. Dann muss Kf den Knoten-Test 'a' erfüllen, auf derselben Ebene wie Kp liegen, und f muss zwischen den Positionen p und $endOfParent_B$(p) liegen, also p<f<$endOfParent_B$(p).

Definition 4.17 ($following-sibling_\sigma(p)$)**.** Sei Σ die Menge aller Label und sei $\sigma \in \Sigma$ ein beliebiges Label. Sei B=(b_1, ..., b_n) ein Bitstrom, sei p eine Position in B und sei IL_σ die invertierte Labelliste zu σ. Dann ist die Menge $following-sibling_\sigma(p) \subseteq IL_\sigma$ definiert als:
$following-sibling_\sigma$(p) :=
{f $\in IL_\sigma$ | $level_B$(p)-$level_B$(f)=0 $\wedge$ p<f<$endOfParent_B$(p)}

Satz 4.11. Seien S ein Struktur-Strom, i und j die Positionen je eines startElement-Events eines Knotens k_i bzw. k_j im Struktur-Strom und B(S)=(b_1, ..., b_n) der Bitstrom von S. Dann gilt:
$k_j \in k_i$/following-sibling::a $\Leftrightarrow$ j $\in following-sibling_a$(i).

Beweis. Beweis folgt aus Definitionen 2.5(g) und 4.17. □

4.2.4 following::a

Sei Kp der aktuelle Kontextknoten und p die Position des '1'-Bits, das Kp im Bitstrom repräsentiert. Sei weiterhin Kf $\in$ Kp/following ein following-Knoten von Kp und f die Position des '1'-Bits, das Kf im Bitstrom B repräsentiert.

Dann muss Kf den Knoten-Test 'a' erfüllen, und f muss hinter der Position end_B(p) liegen, also end_B(p)<f. Es existiert keine Bedingung an $level_B$(f).

Definition 4.18 ($following_\sigma(p)$)**.** Sei Σ die Menge aller Label und sei $\sigma \in \Sigma$ ein beliebiges Label. Sei B=($b_1, \ldots, b_n$) ein Bitstrom, p eine Position in B und sei IL_σ die invertierte Labelliste zu σ. Dann ist die Menge $following_\sigma(p) \subseteq IL_\sigma$ definiert als:
$following_\sigma$(p) :={f $\in IL_\sigma$ | end_B(p)<f<n}

Satz 4.12. Seien S ein Struktur-Strom, i und j die Positionen je eines startElement-Events eines Knotens k_i bzw. k_j im Struktur-Strom und B(S)=($b_1, \ldots, b_n$) der Bitstrom von S. Dann gilt:
$k_j \in k_i$/following::a $\Leftrightarrow$ j $\in following_a$(i).

Beweis. Beweis folgt aus Definitionen 2.5(i) und 4.18. □

Satz 4.13. Sei Σ die Menge aller Label, sei S ein Struktur-Strom, sei B(S)=($b_1, \ldots, b_n$) der Bitstrom von S und IL_σ die invertierte Labelliste zu $\sigma \in \Sigma$. Die Mengen $child_\sigma, descendant_\sigma, following - sibling_\sigma$ und $following_\sigma$ können in O(n) berechnet werden.

Beweis. Die Definition aller 4 Mengen sind von der Form (level-Bedingung $\wedge$ start<x<end), wobei start und end jeweils auch auf Berechnungen der Funktion level zurückführbar sind. Daher reicht zur Berechnung dieser Mengen ein lineares Durchqueren des Bitstroms beginnend an Position start und endend an Position end, wobei für jede Position p mit start<p<end die Level-Differenz $level_B$(start)-$level_B$(p) berechnet wird. Gilt p $\in IL_\sigma$, so wird zusätzlich die level-Bedingung überprüft. Somit reicht ein lineares Durchqueren des Bitstroms im worst case, es gilt also, dass die Mengen in O(n) berechenbar sind. □

Laut den Sätzen 4.9, 4.10, 4.11, 4.12 und 4.13, ermöglichen die Mengen $child_\sigma, descendant_\sigma, following - sibling_\sigma$ und $following_\sigma$ eine effiziente Anfrage-Auswertung direkt auf der Succinct-Darstellung ohne vorherige Dekompression. Dabei können die Anfragen so ausgewertet werden, dass ein einmaliges lineares Durchqueren des Bitstroms genügt; diese Anfrage-Auswertung ist auch für quasi-unendliche, komprimierte Datenströme geeignet.

4.3 Succinct-Darstellung zur Kompression unendlicher Datenströme

Die Succinct-Darstellung, bestehend aus Bitstrom, Labelliste und Symboltabelle, ist uneingeschränkt auf unendliche Struktur-Ströme anwendbar, da je-

weils jedes Event des Struktur-Stroms autonom von allen anderen Events verarbeitet und komprimiert wird.

Für die Succinct-Darstellung bestehend, aus Bitstrom und invertierten Labellisten, gilt zwar auch, dass jedes Event des Struktur-Stroms autonom komprimiert wird, hier tritt jedoch das Problem auf, dass mit steigender Länge des verarbeiteten Strom-Anteils die Größe der Positionen in den invertierten Labellisten ansteigt. Zwar erlaubt die in Kapitel 2.6 vorgestellte Kodierung ganzzahliger Werte auch die Darstellung entsprechend großer Zahlen, dennoch würde die Kompression dadurch ineffizient werden, da große Zahlen auch durch eine entsprechend hohe Anzahl an Bits kodiert werden müssen. Daher empfiehlt es sich, in den invertierten Labellisten nicht die absoluten Positionen zu speichern, sondern jeweils nur die relativen Positionen, also die Differenz zum letztmaligen Auftreten des Labels.

Definition 4.19 (Relative invertierte Labellisten)**.** Sei Σ die Menge aller Label. Sei $\sigma \in \Sigma$ ein beliebiges Label. Sei S=(s_1, ..., s_n) ein Struktur-Strom, und sei IL_σ=($il_{\sigma 1}$, ..., $il_{\sigma m}$) die invertierte Labelliste zu einem Label $\sigma \in \Sigma$. Die umkehrbare Funktion relLL: {1, ..., m} $\rightarrow$ {1, ..., n} ist definiert als

$$relLL_\sigma(\mathrm{x}):=\begin{cases} il_{\sigma 1} & falls\ x = 1 \\ il_{\sigma x} - il_{\sigma x-1} & sonst \end{cases}$$

Die Folge RL_σ=($relLL_\sigma$(1), ..., $relLL_\sigma$(m)) zu einem Label $\sigma \in \Sigma$ bezeichnen wir als *relative invertierte Labelliste zu* σ. Die Menge RL_Σ= $\{RL_\sigma \mid \sigma \in \Sigma\}$ bezeichnen wir als die *relativen invertierten Labellisten zu* Σ *und S.*

Beispiel 4.5 *Tabelle 4.3 zeigt die relativen invertierten Labellisten, die aus dem Struktur-Strom aus Listing 3.1 generiert werden.*

$IL_{@modell}$	(41)
$IL_{Adressen}$	(2)
IL_{Name}	(4, 10, 18)
IL_{Ort}	(22)
IL_{Person}	(3, 10, 18)
$IL_{Postfach}$	(26, 10)
$IL_{Strasse}$	(18)
$IL_{Telefon}$	(40)

Tabelle 4.3: Relative invertierte Labellisten des Beispiels

Die Kompression eines unendlichen Datenstroms durch das Succinct-Verfahren kann ohne weiteren Speicherbedarf für beide Versionen (Bitstrom, Symboltabelle, Labelliste bzw. Bitstrom, invertierte relative Labellisten) durchgeführt werden. Da jedoch der Empfänger – z.B. die Dekompression oder die Anfrage-Auswertung – die aktuellen Werte aus den jeweiligen Komponenten gleichzeitig

benötigt, müssten entsprechend viele Verbindungen von der Kompression zum Empfänger vorhanden sein. Im Falle der Kombination Bitstrom und invertierte Labellisten müssten dies eine Verbindung für den Bitstrom und jeweils eine Verbindung je invertierter Labelliste sein.

Da dies im Allgemeinen nicht praktikabel ist, empfiehlt sich das Aufteilen des Struktur-Stroms in mehrere Pakete anhand eines Parameters, der die maximale Anzahl von Struktur-Events je Struktur-Paket vorgibt. Die verschiedenen Ströme können dann so gebündelt werden, dass die einzelnen Komponenten (Symbol-Strom, invertierte Labelliste und Bitstrom) in jedem Paket nacheinander gesendet werden. Hierbei ist eine sinnvolle Reihenfolge: Symbol-Strom bzw. invertierte Labellisten, Bitstrom. Dies ermöglicht bei der Dekompression, dass die invertierten Labellisten noch vor der Dekompression, die durch den Bitstrom gesteuert wird, in einen Symbol-Strom umgewandelt werden können. Auch für die optimierte Anfrageauswertung von Anfragen der Form achse::a wird die invertierte Labellisten zu 'a' als erstes benötigt.

Durch das Aufteilen in Pakete und das Bündeln der Ströme erreichen wir eine vollständige Streamingfähigkeit des Verfahrens.

4.4 Unterstützung der DOM-Schnittstelle

Um zu zeigen, dass die Succinct-Darstellung die komplette DOM-Schnittstelle unterstützt, werde ich in diesem Kapitel noch erläutern, wie die lesende DOM-Funktion parent sowie die schreibenden DOM-Operationen insertBefore, insertAfter und remove direkt auf dem Komprimat – ohne vorherige Dekompression – definiert sind.

4.4.1 Die parent-Achse

Sei Kc der aktuelle Kontextknoten und c die Position des '1'-Bits, das Kc im Bitstrom B repräsentiert. Dann gilt für den parent-Knoten Kp von Kc, der durch das '1'-Bit an Position p im Bitstrom repräsentiert ist, dass das Level von Kp um eins geringer ist als das Level von Kc, dass p vor c im Bitstrom liegt, und dass kein Knoten existiert, dessen Level um eins geringer ist als das Level von Kc und der zwischen p und c im Bitstrom liegt.

Definition 4.20 (Funktion parent)**.** Sei B=($b_1, \ldots, b_n$) ein korrekter Bitstrom mit der Menge e(B) der Eins-Positionen. Dann ist die Funktion *parent*: e(B)-{1} $\rightarrow$ e(B) definiert als:

parent(x) :=y mit y<x $\wedge$ $level_B(y)$-$level_B(x)$=1 $\wedge$ *not* $\exists$ z: $level_B(z)$-$level_B(x)$ =1 $\wedge$ y<z<x.

Satz 4.14. Sei S ein Struktur-Strom und seien i und j die Positionen je eines startElement-Events eines Knotens k_i bzw. k_j im Struktur-Strom und B(S)=($b_1, \ldots, b_n$) der Bitstrom von S. Dann gilt: k_j=k_i/parent $\Leftrightarrow$ parent(i)=j.

Beweis. Beweis folgt aus Definitionen 2.5(b), 4.1 und 4.20. □

4.4.2 Einfügen und Löschen in Bitstrom, Symbol-Strom und invertierten Labellisten

Die nachfolgende Funktion fügt einen Bitstrom B_{neu} an einer gegebenen Position p in einen anderen Bitstrom B_{alt} ein.

Definition 4.21 ($insert_{B_{alt},B_{neu},p}$)**.** Seien B_{alt}=($b_{alt1}, \ldots, b_{altn}$) und B_{neu}=($b_{neu1}, \ldots, b_{neum}$) zwei Bitströme. Sei p $\in$ {1, ..., n} eine Position in B_{alt}. Dann ist die Funktion $insert_{B_{alt},B_{neu},p}$: {1, ..., n+m} $\rightarrow$ {0,1} definiert als

$$insert_{B_{alt},B_{neu},p}(x):=\begin{cases} b_{altx} & falls\ x < p \\ b_{neu(x-p+1)} & falls\ p \leq x < p+m \\ b_{alt(x-m)} & sonst \end{cases}$$

Entsprechend löscht die remove-Funktion einen Teilbaum bestehend aus l Bits aus dem Bitstrom, dessen Wurzel durch eine Eins-Position p im Bitstrom gegeben ist.

Definition 4.22 ($remove_{B,p}$)**.** Sei B=($b_1, \ldots, b_n$) ein korrekter Bitstrom und sei p $\in$ {1, ..., n} eine Eins-Position in B und sei l:=end(p)-p+1. Dann ist die Funktion $remove_{B,p}$: {1, ..., n-l} $\rightarrow$ {0,1} definiert als

$$remove_{B,p}(x):=\begin{cases} b_x & falls\ x < p \\ b_{x+l} & sonst \end{cases}$$

Die entsprechende insert-Funktion für den Symbol-Strom fügt einen Symbol-Strom SS_{neu} an einer gegebenen Position p in einen anderen Symbol-Strom SS_{alt} ein.

Definition 4.23 ($insert_{SS_{alt},SS_{neu},p}$)**.** Seien SS_{alt}=($ss_{alt1}, \ldots, ss_{altn}$) und SS_{neu}=($ss_{neu1}, \ldots, ss_{neum}$) zwei Symbol-Ströme. Sei p $\in$ {1, ..., n} eine Position in SS_{alt}. Dann ist die Funktion $insert_{SS_{alt},SS_{neu},p}$: {1, ..., n+m} $\rightarrow$ {0,1} definiert als

$$insert_{SS_{alt},SS_{neu},p}(x):=\begin{cases} ss_{altx} & falls\ x < p \\ ss_{neu(x-p+1)} & falls\ p \leq x < p+m \\ ss_{alt(x-m)} & sonst \end{cases}$$

Die folgende remove-Funktion löscht ein Fragment beginnend an Position p der Länge l aus dem Symbol-Strom.

Definition 4.24 ($remove_{SS,p}$)**.** Sei SS=(ss_1, ..., ss_n) ein Symbol-Strom und sei p $\in \{1, \ldots, n\}$ eine Position in B und sei l $\in \{1,\ldots,n\}$. Dann ist die Funktion $remove_{SS,p,l}$: {1, ..., n-l} $\rightarrow$ {0,1} definiert als

$$remove_{SS,p,l}(x) := \begin{cases} ss_x & falls\ x < p \\ ss_{x+l} & sonst \end{cases}$$

Die nachfolgende Funktion fügt eine invertierte Labelliste IL_{neu} in eine andere invertierte Labelliste IL_{alt} ein, so dass alle Werte von IL_{neu} in IL_{alt} zwischen die Werte p (das der Start-Position eines Teilbaumes T in einem zu IL_{alt} gehörenden Bitstrom B_{alt} entspricht) und p+1 eingefügt werden. Die Länge l entspricht hierbei der Länge des zu IL_{neu} gehörenden Bitstroms B_{neu}. Dies erfordert eine Neuberechnung aller Werte, die größer als p sind.

Definition 4.25 ($insert_{IL_{alt},IL_{neu},p}$)**.** Seien IL_{alt}=(il_{alt1}, ..., il_{altn}) und IL_{neu} =(il_{neu1}, ..., il_{neum}) zwei invertierte Labellisten. Seien p und l zwei Integer-Werte und sei k:=$|\{il_{altx} \mid il_{altx} < p\}|$. Dann ist die Funktion $insert_{IL_{alt},IL_{neu},p,l}$: {1, ..., n+m} $\rightarrow$ Integer definiert als

$$insert_{IL_{alt},IL_{neu},p,l}(x) := \begin{cases} il_{altx} & falls\ x \leq k \\ il_{neu(x-k)} + p & falls\ k < x \leq k+m \\ il_{alt(x-m)} + l & sonst \end{cases}$$

Die folgende remove-Funktion löscht alle Positionen x mit p≤x≤p+l für gegebenen Werte p und l aus einer invertierten Labelliste.

Definition 4.26 ($remove_{IL,p,l}$)**.** Sei IL=(il_1, ..., il_n) eine invertierte Labelliste. Seien p, l zwei Integer und sei k:=$|\{il_x \mid il_x < p\}|$ und o:=$|\{il_x \mid p \leq il_x \leq p+l\}|$. Dann ist die Funktion $remove_{IL,p,l}$: {1, ..., o-p} $\rightarrow$ Integer definiert als

$$remove_{IL,p,l}(x) := \begin{cases} il_x & falls\ x \leq k \\ il_{(x+o)} - l & sonst \end{cases}$$

4.4.3 insert und remove

Die folgenden Sätze zeigen, dass eine Ausführung der insert- bzw. der remove-Funktion auf dem Struktur-Strom S zu dem gleichen Ergebnis führt wie ein Ausführen der insert- bzw. remove-Funktion auf dem S entsprechenden Komprimat bestehend aus Bitstrom und invertierten Labellisten zuzüglich anschließender Dekompression.

Satz 4.15. Seien SS_{alt}=(ss_{alt1}, ..., ss_{altn}) und SS_{neu}=(ss_{neu1}, ..., ss_{neum}) zwei Symbol-Ströme. Sei p mit 1≤p≤n gegeben. Sei SS=(ss_1, ..., ss_{m+n}):= ($insert_{SS_{alt},SS_{neu},p}(1)$, ..., $insert_{SS_{alt},SS_{neu},p}(m+n)$). Sei Σ die Menge aller Label, und seien $IL_{\Sigma alt}$=$\{IL_{\sigma alt} | \sigma \in \Sigma\}$ mit $IL_{\sigma alt}$=($il_{\sigma alt1}$, ..., $il_{\sigma altk}$) und

$IL_{\Sigma neu}$=$\{IL_{\sigma neu}|\sigma \in \Sigma\}$ mit $IL_{\sigma neu}$=($il_{\sigma neu1}$, ..., $il_{\sigma neul}$) zwei Mengen von invertierten Labellisten. Sei IL_{Σ}:=$\{IL_{\sigma}|\sigma \in \Sigma\}$ mit

IL_{σ}=($insert_{IL_{\sigma alt},IL_{neu\sigma},p,m}(1)$, ..., $insert_{IL_{\sigma alt},IL_{neu\sigma},p,m}(k+l)$).

Dann gilt: $invToSS_{IL_{\Sigma}alt}$(i)=ss_{alti} für 1≤i≤n und $invToSS_{IL_{\Sigma}neu}$(j)=ss_{neuj} für 1≤j≤m ⇔ $invToSS_{IL_{\Sigma}}$(x)=ss_x für 1≤x≤n+m.

Beweis. Folgt aus Definitionen 4.14, 4.23 und 4.25. □

Satz 4.16. Sei SS_{alt}=(ss_{alt1}, ..., ss_{altn}) ein Symbol-Strom. Seien p,l mit 1≤p,l≤n gegeben. Sei SS=(ss_1, ..., ss_{n-l}):=($remove_{SS_{alt},p,l}(1)$, ..., $remove_{SS_{alt},p,l}(n-l)$). Sei Σ die Menge aller Label, und sei $IL_{\Sigma alt}$=$\{IL_{\sigma alt}|\sigma \in \Sigma\}$ mit $IL_{\sigma alt}$=($il_{\sigma alt1}$, ..., $il_{\sigma altk}$) eine Menge von invertierten Labellisten. Sei IL_{Σ}:=$\{IL_{\sigma}|\sigma \in \Sigma\}$ mit IL_{σ}=($remove_{IL_{\sigma alt},p,l}(1)$, ..., $remove_{IL_{\sigma alt},p,l}(k-l)$).

Dann gilt: $invToSS_{IL_{\Sigma}alt}$(i)=ss_{alti} für 1≤i≤n ⇔ $invToSS_{IL_{\Sigma}}$(x)=ss_x für 1≤x≤n-l.

Beweis. Folgt aus Definitionen 4.14, 4.24 und 4.26. □

Satz 4.17. Seien S_{alt}=(s_{alt1}, ..., s_{altn}) und S_{neu}=(s_{neu1}, ..., s_{neum}) zwei Struktur-Ströme. Sei p mit 1≤p≤n gegeben. Sei S=(s_1, ..., s_{m+n}):=($insert_{S_{alt},S_{neu},p}(1)$, ..., $insert_{S_{alt},S_{neu},p}(m+n)$). Seien weiterhin B_{alt}=(b_{alt1}, ..., b_{altn}) und B_{neu}=(b_{neu1}, ..., b_{neum}) zwei Bitströme. Sei B=(b_1, ..., b_{m+n}):=($insert_{B_{alt},B_{neu},p}(1)$, ..., $insert_{B_{alt},B_{neu},p}(m+n)$). Seien SS_{alt}=(ss_{alt1}, ..., $ss_{alt\frac{n}{2}}$) und SS_{neu}=(ss_{neu1}, ..., $sS_{neu\frac{m}{2}}$) zwei Symbol-Ströme. Sei

SS=(ss_1, ..., $ss_{\frac{m+n}{2}}$):=($insert_{SS_{alt},SS_{neu},p}(1)$, ..., $insert_{SS_{alt},SS_{neu},p}(\frac{m+n}{2})$).

Dann gilt $decomp_{SS_{alt},B_{alt}}$(i)=$ss_{alti}$ für 1≤i≤n ⇔ $decomp_{SS,B}$(x)=ss_x für 1≤x≤m+n.

Beweis. Folgt aus Definitionen 2.7, 4.10, 4.21 und 4.23. □

Satz 4.18. Sei S_{alt}=(s_{alt1}, ..., s_{altn}) ein Struktur-Strom. Sei p mit 1≤p≤n gegeben. Sei ss_p das startElement-Event eines XML-Elements E zu einem gegebenen Wert p und sei ss_k das endElement-Event von E. Sei l:=k-p+1. Sei S=(s_1, ..., s_{n-l}):=($remove_{S_{alt},p}(1)$, ..., $remove_{S_{alt},p}(n-l)$). Sei weiterhin B_{alt}=(b_{alt1}, ..., b_{altn}) ein Bitstrom. Sei B=(b_1, ..., b_{n-l}):=($remove_{B_{alt},p}(1)$, ..., $remove_{B_{alt},p}(n-l)$). Sei SS_{alt}=(ss_{alt1}, ..., $ss_{alt\frac{n}{2}}$) ein Symbol-Strom. Sei SS=(ss_1, ..., $ss_{\frac{n-l}{2}}$):=($remove_{SS_{alt}p,l}(1)$, ..., $remove_{SS_{alt},p,l}(\frac{n-l}{2})$).

Dann gilt $decomp_{SS_{alt},B_{alt}}$(i)=$ss_{alti}$ für 1≤i≤n ⇔ $decomp_{SS,B}$(x)=ss_x für 1≤x≤n-l.

Beweis. Folgt aus Definitionen 2.8, 4.10, 4.22 und 4.24. □

Satz 4.19. Sei Σ die Menge aller Label, sei S ein Struktur-Strom, sei B(S)=(b_1, ..., b_n) der Bitstrom von S und IL_σ die invertierte Labelliste zu $\sigma \in \Sigma$. Die Funktionen insert und remove für B,S und IL können in O(n) berechnet werden.

Beweis. Da entsprechend der Definitionen all diese Funktionen durch ein einmaliges Durchqueren der jeweiligen Folge berechnet werden können, sind sie in O(n) berechenbar. □

Laut den Sätzen 4.15, 4.16, 4.17, 4.18 und 4.19 ermöglichen die Funktionen insert und remove für Bitstrom, Symbol-Strom und invertierte Labellisten eine effiziente Implementierung der DOM-Schnittstelle direkt auf der Succinct-Darstellung ohne vorherige Dekompression. Dabei können die Update-Operationen so ausgeführt werden, dass ein einmaliges lineares Durchqueren der jeweiligen Struktur genügt. Dies gilt sowohl für eine Succinct-Darstellung bestehend aus Bitstrom und Symbol-Strom, als auch für eine Succinct-Darstellung bestehend aus Bitstrom und invertierten Labellisten.

4.5 Zusammenfassung: Eigenschaften der Succinct-Darstellung

4.5.1 Kompressionsstärke

Betrachten wir die Baumdarstellung eines XML-Dokumentes, so benötigt die Succinct-Darstellung den folgenden Speicherbedarf für die einzelnen Repräsentationen der Dokument-Struktur. Hierbei gehen wir von der vereinfachten Annahme aus, dass das gesamte Dokument in einem Paket gespeichert werden kann. Seien n die Anzahl Baumknoten, length(name) die Anzahl Zeichen von name und count(name) die Anzahl Knoten, die mit diesem Label existieren:

- *Bitstrom:* 2*n Bits.
- *Symbol-Strom:* Je Element- bzw. Attribut-Name name: length(name) * Char + 1 * Integer für die Symboltabelle zzgl. count(name)*Integer für die Labelliste.
- *(relative) invertierte Labelliste*: Je Element-bzw. Attribut-Name ($\neq$"=T"): length(name) * Char + count(name)*Integer + 1*Integer (für Listenende).

4.5.2 Weitere Eigenschaften

Wie im Verlaufe dieses Kapitels gezeigt, hat die Succinct-Darstellung die folgenden Eigenschaften:

- *Streamingfähig:* Sie ist mit Hilfe diskreter Fenster (Pakete) uneingeschränkt streamingfähig.
- *Auswertung von Pfad-Anfragen:* Pfad-Anfragen können direkt auf dem Komprimat ausgewertet werden.
- *Updates:* Updates können unbeschränkt auf dem Komprimat durchgeführt werden, insofern, als das Komprimat mit darauf ausgeführtem Update identisch dazu ist, dass man erst dekomprimiert hätte, die Updates auf dem XML-Dokument ausgeführt hätte und dann anschließend wieder komprimiert hätte.
- *DOM:* Sie unterstützt die DOM-Schnittstelle.

5 XML-Kompression durch Eliminierung struktureller Redundanzen

Das zweite Kompressions-Verfahren, welches ich in dieser Arbeit vorstellen werde, komprimiert XML-Dokumente durch Zusammenfassen von gleichen Teilbäumen. Ähnliche Verfahren wurden bereits in [25, 28] vorgestellt. Das hier vorgestellte Verfahren unterscheidet sich jedoch von den bisherigen Verfahren darin, dass es einerseits durch eine Überlaufstrategie auf potentiell unendliche Datenströme angewandt werden kann und dass es andererseits einen komprimierten SAX-ähnlichen Strom erzeugen kann. Dieser Ereignis-Strom kann als Eingabe für weitere Kompressoren dienen. Da dieses Verfahren somit aus dem XML-Baum einen Baum zzgl. Rückwärtskanten, also einen DAG (=directed acyclic graph) macht, werde ich im weiteren Verlauf dieser Arbeit das Verfahren als DAG-Verfahren bezeichnen.

5.1 XML-Kompression durch Zusammenfassen von gleichen Teilbäumen

5.1.1 Konzept

Die Grundidee des DAG-Verfahrens ist die Eliminierung struktureller Redundanzen durch das Zusammenfassen von gleichen Teilbäumen. Tritt innerhalb der Struktur ein Teilbaum mit Wurzel w zum wiederholten Male auf, so wird diese Wiederholung gelöscht, und statt dessen wird ein Zeiger vom parent-Knoten von w zum erstmaligen Auftreten von w erzeugt.

Die folgende, rekursive Definition definiert, wann zwei Teilbäume gleich sind:

Definition 5.1 (gleiche Teilbäume)**.** Sei B=(V,E) ein geordneter Baum mit Knotenmenge V und Kantenmenge $E \subseteq V \times V$. Sei zu einem Knoten k k.label das Label von k und k.child(i) der Teilbaum mit dem i-ten Kindknoten von k als Wurzel und k.noOfChildren die Anzahl an Kindknoten des Knotens k.

Zwei Teilbäume tb1=(V1,E1), tb2=(V2,E2) mit V1,V2 $\subseteq$ V und E1,E2 $\subseteq$ E mit Wurzelknoten w1 bzw. w2 sind *gleich*, tb1 $\bowtie$ tb2, genau dann, wenn gilt

- $w1.label = w2.label$
- $w1.noOfChildren = w2.noOfChildren$
- $\forall\ i\ mit\ 1 \leq i \leq w1.noOfChildren$: w1.child(i) $\bowtie$ w2.child(i)

□

Dementsprechend ist der minimale DAG zu einem Baum B ein Graph, der keine zwei gleichen Teilbäume enthält, der jedoch alle Kantenbeziehungen von B enthält:

Definition 5.2 (minimaler DAG)**.** Sei XML=(V_{xml}, E_{xml}) ein geordneter XML-Baum mit Knotenmenge V_{xml} und Kantenmenge $E_{xml} \subseteq V_{xml} \times V_{xml}$. Sei DAG=$(V_{dag}, E_{dag})$ ein azyklischer Graph mit Knotenmenge V_{dag} und Kantenmenge $E_{dag} \subseteq V_{dag} \times V_{dag}$. Dann gilt DAG ist der *minimale DAG* zu XML, genau dann, wenn gilt:

- $\forall x_{xml} \in V_{xml}\ \exists x_{dag} \in V_{dag}\ mit\ x_{xml} \bowtie x_{dag}$
- $\forall x_{dag} \in V_{dag}\ \exists x_{xml} \in V_{xml}\ mit\ x_{dag} \bowtie x_{xml}$
- $\forall (x_{xml}, y_{xml}) \in E_{xml}\ \exists (x_{dag}, y_{dag}) \in E_{dag}$ mit $x_{xml} \bowtie x_{dag}$ und $y_{xml} \bowtie y_{dag}$
- $\forall (x_{dag}, y_{dag}) \in E_{dag}\ \exists (x_{xml}, y_{xml}) \in E_{xml}$ mit $x_{xml} \bowtie x_{dag}$ und $y_{xml} \bowtie y_{dag}$
- $not\ \exists x1, x2 \in V_{dag}$ mit $x1 \neq x2$ und $x1 \bowtie x2$.

□

Beispiel 5.1 *Abbildung 5.1 zeigt den minimalen DAG zum Strukturanteil der binären XML-Baumdarstellung aus Abbildung 2.2.*

Da der DAG die Kantenbeziehungen des XML-Baums erhält, können die navigierenden DOM-Operationen genau wie auf dem ursprünglichen XML-Baum durchgeführt werden.

5.1.2 Vergleich binärer DAG zu herkömmlichem DAG

Das zugrunde liegende Konzept, wie es im vorherigen Abschnitt beschrieben wurde, ist zunächst einmal unabhängig von der Baum-Darstellung – es ist sowohl auf herkömmliche XML-Bäume anwendbar, als auch auf Binärbaum-Darstellungen. Wenden wir dieses Konzept sowohl auf die herkömmliche Baum-Darstellung als auch auf die Binärbaum-Darstellung desselben Dokumentes an, so können wir Folgendes feststellen: Obwohl die beiden Baum-Darstellungen

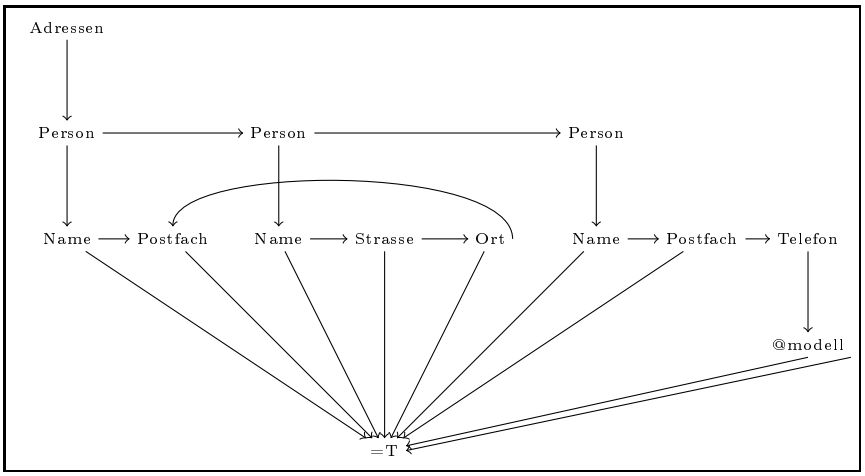

Abbildung 5.1: Minimaler DAG zur Binärbaum-Darstellung des XML-Dokumentes

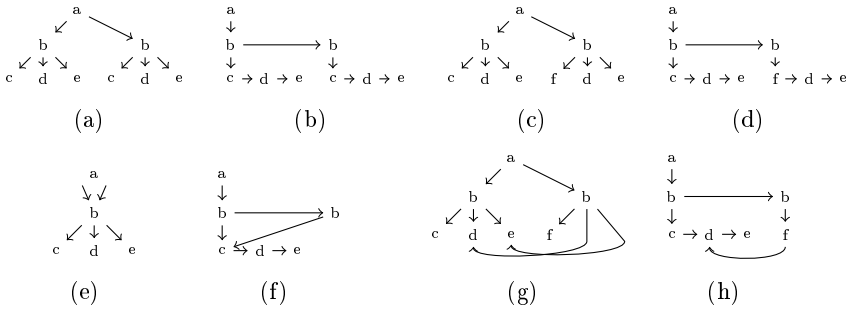

Abbildung 5.2: Vergleich binärer DAG zu herkömmlichem DAG

gleiche Anzahl an Knoten und Kanten besitzen, gilt letzteres im Allgemeinen nicht für die jeweiligen DAG-Darstellungen.

Beispiel 5.2 *Abbildung 5.2(a) zeigt den herkömmlichen XML-Baum und Abbildung 5.2(b) den binären XML-Baum, jeweils darunter die entsprechenden DAGs (Abbildungen 5.2(e) und (f)). Wie wir sehen, hat der herkömmliche DAG in diesem Fall 5 Knoten und 5 Kanten, während der binäre DAG 6 Knoten und 6 Kanten hat, also* größer *ist. Abbildungen 5.2(c),(d),(g) und (h) zeigen dies noch einmal für einen leicht veränderten XML-Baum. Wie wir sehen, hat dieses Mal der herkömmliche DAG 7 Knoten und 8 Kanten, während der binäre DAG 7 Knoten und 7 Kanten hat, also* kleiner *ist.*

Wie an diesem Beispiel zu sehen ist, kann man nicht im Allgemeinen sagen, dass der binäre DAG dem herkömmlichen DAG in Bezug auf Kompressions-

stärke überlegen ist. Da jedoch der binäre DAG einfacher darzustellen ist, da jeder Knoten maximal 2 Kindknoten hat, während im herkömmlichen DAG jeder Knoten eine beliebige Anzahl an Kindknoten haben kann, werde ich im weiteren Verlauf dieses Kapitels das Verfahren auf dem binären DAG erläutern. Es ist aber anzumerken, dass die hier beschriebenen Ideen mit wenigen Änderungen auch auf den herkömmlichen DAG angewandt werden können.

5.1.3 DAG-Event-Strom

Hat man den minimalen DAG zu einem binären XML-Baum berechnet, so kann man – analog wie einen SAX-Strom zu einem XML-Baum – einen DAG-Event-Strom zu diesem DAG generieren, der z.B. als Eingabe für einen weiteren Verarbeitungsschritt dienen kann.

Für diesen DAG-Event-Strom wird als Eigenschaft vorausgesetzt, dass Verweise immer nur "rückwärts" erfolgen, dass also die verwiesenen Knoten immer bereits gelesen wurden, bevor ein Verweis auf diese Knoten erfolgt.

Der DAG-Event-Strom enthält außer den Ereignissen des zugrunde liegenden Stroms noch das DAG-Ereignis pointer(int offset). Das Ereignis pointer(int offset) verweist auf die Ereignissequenz startend bei dem Ereignis, das 'offset' Ereignisse zurückliegt. Das Ereignis pointer(int offset) folgt also 'offset' Ereignisse später als das verwiesene Ereignis.

Beispiel 5.3 *Listing 5.1 enthält den DAG-Strom des DAGs zum Struktur-Strom aus Listing 3.1. Hierbei ist zu beachten, dass der DAG bzgl. des Binärbaums berechnet wurde.*

```
1  startElement('root');
2  startElement('Adressen');
3  startElement('Person');
4  startElement('Name');
5  startElement('=T');
6  endElement   ();
7  endElement   ();
8  startElement('Postfach');
9  pointer(4);   //Verweis auf Zeile 5
10 endElement   ();
11 startElement('Person');
12 startElement('Name');
13 pointer(8);   //Verweis auf Zeile 5
14 startElement('Strasse');
15 pointer(10); //Verweis auf Zeile 5
16 startElement('Ort');
```

```
17 pointer(12); //Verweis auf Zeile 5
18 pointer(10); //Verweis auf Zeile 8
19 startElement('Person');
20 startElement('Name');
21 pointer(16); //Verweis auf Zeile 5
22 startElement('Postfach');
23 pointer(18); //Verweis auf Zeile 5
24 startElement('Telefon');
25 startElement('@modell');
26 pointer(21); //Verweis auf Zeile 5
27 pointer(22); //Verweis auf Zeile 5
28 endElement   ();
29 endElement   ();
```

Listing 5.1: DAG-Event-Strom zu Listing 3.1

Da es sich bei dem DAG-Event-Strom um eine Erweiterung des Struktur-Stroms handelt, handelt es sich beim Struktur-Strom um einen Sonderfall des DAG-Event-Stroms. Alle Anwendungen, die für einen DAG-Event-Strom geschrieben sind, können entsprechend auch auf den zugrunde liegenden Struktur-Strom angewandt werden.

5.1.4 Implementierung einer DAG-Kompression

Nachdem ich das der DAG-Kompression zugrunde liegende Konzept erläutert habe, stelle ich nun eine mögliche Implementierung dieses Konzeptes vor.

Die Eingabe dieser Implementierung ist ein binärer Struktur-Strom, woraus dann eine Liste von DAG-Knoten erzeugt wird. Jeder DAG-Knoten besteht aus einem Label und jeweils einem Zeiger auf den first-child- und den next-sibling-Knoten. Im Unterschied zu einer Baum-Darstellung, bei der jeder Knoten einen Eingangsgrad von maximal 1 hat (nur die Wurzel hat Eingangsgrad 0), kann der Eingangsgrad eines DAG-Knotens beliebig groß werden im Falle einer Verzeigerung auf vorhergehende, sich wiederholende Teilbäume.

In dieser Implementierung wird ein Stack verwendet, der den Pfad von der Wurzel bis zum aktuellen Knoten enthält, da für diese Knoten noch nicht alle Informationen vorhanden sind, next-sibling und für den zuletzt gelesen Knoten evtl. auch first-child eines solchen Knotens sind noch unbekannt.

Algorithmus 5.2 fasst diese Implementierung zusammen. Da die Eingabe der Implementierung ein binärer Struktur-Strom ist, also eine Folge von Events vom Typ firstChild, nextSibling und parent, wird im Folgenden die Aktionen, die bei der Auslösung eines dieser Events ausgeführt wird, erläutert:

1. *firstChild (Zeilen 5-10):* Bislang ist über den aktuellen durch das binäre Event firstChild beschriebenen Knoten nur das Knotenlabel bekannt. Daher wird ein DAG-Knoten mit entsprechendem Label erzeugt (Zeile 6). Um später unterscheiden zu können, ob auf diesen Knoten ein Zeiger vom Typ FirstChild oder NextSibling verweist, wird gespeichert, dass dieser Knoten vom Typ FirstChild ist (Zeile 7) und der Knoten wird zum Stack der noch offenen Elemente hinzugefügt (Zeile 9). Für den Fall, dass es sich bei dem Knoten um den Wurzelknoten handelt, also der Stack derzeit noch leer ist, wird dem DAG dieser Knoten als Wurzelknoten mitgeteilt (Zeile 8).
2. *nextSibling (Zeilen 12-19):* Über den zuletzt geöffneten, aber noch nicht geschlossenen Knoten ist bekannt, dass alle seine Kindknoten bereits gelesen wurden. Falls bis jetzt kein first-child-Knoten für diesen aufgetreten ist, folgt daraus, dass kein first-child-Knoten existiert. Daher wird der first-child-Zeiger des obersten Stack-Elements auf „NOTHING“ gesetzt, falls der Zeiger noch nicht gesetzt ist (Zeilen 13-15). Anschließend wird ein neuer DAG-Knoten mit entsprechendem Label und dem Typ NextSibling erzeugt (Zeilen 16-17) und zum Stack der noch offenen Elemente hinzugefügt (Zeile 18).
3. *parent (Zeilen 21-28):* Über den zuletzt geöffneten, aber noch nicht geschlossenen Knoten ist bekannt, dass alle seine Kindknoten und auch Geschwisterknoten bereits gelesen wurden. Falls bis jetzt kein first-child-Knoten für diesen aufgetreten ist, folgt daher, dass kein first-child-Knoten existiert (Zeilen 23-24). Ebenso folgt, dass kein next-sibling-Knoten existiert, falls bis jetzt noch kein next-sibling-Knoten aufgetreten ist (Zeilen 25-26). Da nun alle Daten des obersten DAG-Knotens bekannt sind, kann dieser vom Stack in den DAG verschoben werden. Dort wird überprüft, ob bereits ein gleicher Knoten exisitiert, und wird entweder die entsprechende ID zurückgeliefert, oder eine neu erzeugte ID (Zeile 33). Dies wird durch die ausgelagerte Methode clearStack übernommen. Wir verdrängen das in den DAG eingefügte Element lastElement vom Stack (Zeile 34), und wir setzen je nach Typ von lastElement entweder den first-child- oder den next-sibling-Zeiger des nun oben auf dem Stack befindenden Knotens auf lastElement (Zeilen 35-38), somit ist der Zeiger vom parent-Knoten bzw. vom previous-sibling-Knoten entsprechend der im XML-Baum bestehenden Beziehung korrekt gesetzt. Dies wird solange wiederholt, wie noch vollständige DAG-Elemente auf dem Stack liegen.
   Das Einfügen von Knoten in den DAG und das Überprüfen, ob ein DAG-Knoten bereits vorhanden ist, kann effizient mit Hilfe eines Hashings realisiert werden.

Definition 5.3 (BinarySAX2DAG)**.** Sei BS ein binärer Struktur-Strom. Dann ist die Klasse BinarySAX2DAG mit den BinarySAX-Operationen firstChild(String name), nextSibling(String name) und parent() von BS wie folgt definiert:

```
1  public class BinarySAX2DAG implements BinarySAX{
2     private Stack openElements;
3     public DAG dag;
4
5     public void firstChild(String name){
6        DAGEntry fc = new DAGEntry(name);
7        fc.setType(DAGEntry.FirstChild);
8        if(openElements.isEmpty())DAG.setRoot(fc);
9        openElements.push(fc);
10    }
11
12    public void nextSibling(String name){
13       DAGEntry lastElement = openElements.top();
14       if(lastElement.getFC()==null)
15          lastElement.setFC(DAGEntry.NOTHING);
16       DAGEntry ns = new DAGEntry(name);
17       ns.setType(DAGEntry.NextSibling);
18       openElements.push(ns);
19    }
20
21    public void parent(){
22       DAGEntry lastElement = openElements.top();
23       if(lastElement.getFC()==null)
24          lastElement.setFC(DAGEntry.NOTHING);
25       if(lastElement.getNS()==null)
26          lastElement.setNS(DAGEntry.NOTHING);
27       clearStack();
28    }
29
30    private void clearStack(){
31       DAGEntry lastElement = openElements.top();
32       while(lastElement.getFC()!=null && lastElement.
            getNS()!=null){
33          int ID = DAG.insert(lastElement);
34          openElements.pop();
35          if(lastElement.getType()==DAGEntry.FirstChild)
```

```
36                openElements.top().setFC(ID);
37            else
38                openElements.top().setNS(ID);
39            lastElement = openElements.top();
40        }
41    }
42 }
```

Algorithmus 5.2: DAG-Kompression des Struktur-Stroms

Beispiel 5.4 *Tabelle 5.1 zeigt den DAG zu Listing 2.3. Hierbei ist der mit '→' markierte DAG-Knoten der Wurzelknoten und '-' steht für DAGEntry.NOTHING.*

ID	Label	First-Child	Next-Sibling
1	=T	-	-
2	Postfach	1	-
3	Name	1	2
4	Ort	1	2
5	Strasse	1	4
6	Name	1	5
7	@modell	1	1
8	Telefon	7	-
9	Postfach	1	8
10	Name	1	9
11	Person	10	-
12	Person	6	11
13	Person	3	12
→14	Adressen	13	-

Tabelle 5.1: DAG zum Beispieldokument aus Listing 2.3

Dadurch, dass neue DAG-Einträge erst beim parent-Event des binären SAX-Event-Stroms vom Stack in den DAG verschoben werden, der DAG also bottom-up aufgebaut wird, hat der durch diese Implementierung erzeugte DAG die Eigenschaft, dass Verweise immer nur "rückwärts" erfolgen, dass also die verwiesenen Knoten immer bereits gelesen wurden, bevor ein Verweis auf diese Knoten erfolgt.

5.1.5 Implementierung einer DAG-Dekompression

Algorithmus 5.3 enthält ein DAG-Objekt, also eine Liste von DAG-Knoten und erzeugt daraus einen binären Struktur-Strom.

Dazu wird die DAG-Knoten-Liste rekursiv durchquert, startend mit dem Wurzelknoten des DAGs (Zeile 5). Je nach Typ des aktuellen DAG-Knotens wird ein firstChild- oder ein nextSibling-Event des binären Sturktur-Stroms binarySAX erzeugt (Zeilen 9-12). Falls ein first-child-Zeiger existiert, wird anschließend die Dekompression für diesen Knoten aufgerufen (Zeilen 13-14). Ebenso wird die Dekompression für den next-sibling aufgerufen, falls dieser existiert (Zeilen 15-16). Existiert kein next-sibling, so bedeutet dies, dass das Ende der sibling-Liste erreicht ist, somit wird ein parent-Event erzeugt (Zeilen 17-18).

Definition 5.4 (decompress)**.** Sei dag ein DAG. Dann ist die Operation decompress(DAG dag) in der folgenden Funktion decompress definiert:

```
1  public class DAG2BinarySAX{
2      BinarySAX binarySAX;
3
4      public void decompress(DAG dag){
5          decompressNode(dag.getRoot(),DAGEntry.FirstChild);
6      }
7
8      public void decompressNode(DAGEntry node, Type type){
9          if(type==DAGEntry.FirstChild)
10             binarySAX.firstChild(node.getLabel());
11         else
12             binarySAX.nextSibling(node.getLabel());
13         if(node.getFirstChild()!=DAGEntry.NOTHING)
14           decompressNode(node.getFirstChild(), DAGEntry.
                 FirstChild);
15         if(node.getNextSibling()!=DAGEntry.NOTHING)
16           decompressNode(node.getNextSibling(), DAGEntry.
                 NextSibling);
17         else
18             binarySAX.parent();
19     }
20 }
```

Algorithmus 5.3: Dekompression der DAG-Kompression

Der folgende Satz besagt, dass es sich bei den vorgestellten Kompressions- und Dekompressions-Operationen um eine korrekte Kompression handelt, dass also die Dekompression wieder den ursprünglichen Struktur-Strom herstellt. Dies gilt nicht nur für den gesamten binären Struktur-Strom, sondern für alle

im binären Struktur-Strom enthaltenen vollständigen binären Teilbäume. Bevor der Satz vorgestellt wird, muss zunächst einmal ein vollständiger binärer Teilbaum definiert werden:

Definition 5.5 (Vollständiger binärer Teilbaum)**.** Sei BS=(bs_1, ..., bs_n) ein binärer Struktur-Strom nach Definition 3.3. Seien fc_x:=$\{bs_i \in BS | 1 \leq i \leq x, bs_i$ ist vom Typ firstChild$\}$, ns_x:=$\{bs_i \in BS | 1 \leq i \leq x, bs_i$ ist vom Typ nextSibling$\}$ und pa_x:=$\{bs_i \in BS | 1 \leq i \leq x, bs_i$ ist vom Typ parent$\}$. Dann bezeichnen wir BS als *vollständigen binären Teilbaum* genau dann, wenn gilt:

1. $bs_1 \in fc_1 \wedge |pa_n| = |fc_n| \wedge \forall x, 1 \leq x < n : |pa_x| < |fc_x|$ oder
2. $bs_1 \in ns_1 \wedge |pa_n| = |fc_n| + 1 \wedge \forall x, 1 \leq x < n : |pa_x| < |fc_x| + 1$

Satz 5.1. Sei BS ein binärer Struktur-Strom nach Definition 3.3, und sei vBS $\subseteq$ BS ein vollständiger binärer Teilbaum nach Definition 5.5. Sei dag der DAG, der entsteht, wenn die in der Klasse BinarySAX2DAG aus Definition 5.3 definierten Aktionen für die binären SAX-Events von vBS ausgeführt werden. Sei decompress(DAG dag) eine Dekompressions-Funktion entsprechend Definition 5.4. Dann gilt: decompress(dag) erzeugt vBS.

Beweis. Sei 'x' der Elementname des Wurzelknotens von vBS, 'y' der Elementname des parents des Wurzelknotens von vBS.

- Der kleinste vollständige binäre Teilbaum vBS ist ein Blattknoten, ist also entweder von der Form firstChild('x') $\otimes$ parent() (Fall 1a) oder nextSibling('x') $\otimes$ parent() (Fall 1b).
  - Fall 1a: Der resultierende DAG enthält die DAG-Knoten D1=(a, y, b, UNKNOWN) und D2=(b, x, NOTHING, NOTHING), wobei a die ID von D1 und b die ID von D2 ist. Anwendung der Operation decompressNode auf den Knoten D1 führt somit zu einem Aufruf der Operation decompressNode(D2, DAGEntry.FirstChild). Diese erzeugt die Folge firstChild('x') $\otimes$ parent().
  - Fall 1b: Der resultierende DAG enthält die DAG-Knoten D1=(a, y, UNKNOWN, b) und D2=(b, x, NOTHING, NOTHING), wobei a die ID von D1 und b die ID von D2 ist. Anwendung der Operation decompressNode auf den Knoten D1 führt somit zu einem Aufruf der Operation decompressNode(D2, DAGEntry.NextSibling). Diese erzeugt die Folge nextSibling('x') $\otimes$ parent().
- Nun bestehe der vollständige binäre Teilbaum vBS aus einem Wurzelknoten und einem first-child, es existiere aber kein next-sibling des Wurzelknotens. Dann ist vBS von der Form firstChild('x') $\otimes$ vFC $\otimes$ parent() (Fall 2a) oder nextSibling('x') $\otimes$ vFC $\otimes$ parent() (Fall 2b), wobei für den

vollständigen binären Teilbaum vFC mit Wurzel vom Typ firstChild die Behauptung gelte, so dass fc die DAG-ID des Wurzelknotens von vFC sei.

  - Fall 2a: Der resultierende DAG enthält die DAG-Knoten D1=(a, y, b, UNKNOWN) und D2=(b, x, fc, NOTHING), wobei a die ID von D1 und b die ID von D2 ist. Anwendung der Operation decompressNode auf den Knoten D1 führt somit zu einem Aufruf der Operation decompressNode(D2, DAGEntry.FirstChild). Diese erzeugt die Folge firstChild('x') $\otimes$ vFC $\otimes$ parent().
  - Fall 2b: Der resultierende DAG enthält die DAG-Knoten D1=(a, y, UNKNOWN, b) und D2=(b, x, fc, NOTHING), wobei a die ID von D1 und b die ID von D2 ist. Anwendung der Operation decompressNode auf den Knoten D1 führt somit zu einem Aufruf der Operation decompressNode(D2, DAGEntry.NextSibling). Diese erzeugt die Folge nextSibling('x') $\otimes$ vFC $\otimes$ parent().

- Nun bestehe der vollständige binäre Teilbaum vBS aus einem Wurzelknoten und einem next-sibling, es existiere aber kein first-child des Wurzelknotens. Dann ist vBS von der Form firstChild('x') $\otimes$ vNS (Fall 3a) oder nextSibling('x') $\otimes$ vNS (Fall 3b), wobei für den vollständigen binären Teilbaum vNS mit Wurzel vom Typ nextSibling die Behauptung gelte, so dass ns die DAG-ID des Wurzelknotens von vNS sei.

  - Fall 3a: Der resultierende DAG enthält die DAG-Knoten D1=(a, y, b, UNKNOWN) und D2=(b, x, NOTHING, ns), wobei a die ID von D1 und b die ID von D2 ist. Anwendung der Operation decompressNode auf den Knoten D1 führt somit zu einem Aufruf der Operation decompressNode(D2, DAGEntry.FirstChild). Diese erzeugt die Folge firstChild('x') $\otimes$ vNS.
  - Fall 3b: Der resultierende DAG enthält die DAG-Knoten D1=(a, y, UNKNOWN, b) und D2=(b, x, NOTHING, ns), wobei a die ID von D1 und b die ID von D2 ist. Anwendung der Operation decompressNode auf den Knoten D1 führt somit zu einem Aufruf der Operation decompressNode(D2, DAGEntry.NextSibling). Diese erzeugt die Folge nextSibling('x') $\otimes$ vNS.

- Schließlich bestehe der vollständige binäre Teilbaum vBS aus einem Wurzelknoten, einem first-child und einem next-sibling des Wurzelknotens. Dann ist vBS von der Form firstChild('x') $\otimes$ vFC $\otimes$ vNS (Fall 4a) oder nextSibling('x') $\otimes$ vFC $\otimes$ vNS (Fall 4b), wobei für die vollständigen binären Teilbäume vFC mit Wurzel vom Typ firstChild und vNS mit Wurzel vom Typ nextSibling die Behauptung gelte, so dass fc die DAG-

ID des Wurzelknotens von vFC und ns die DAG-ID des Wurzelknotens von vNS seien.

- Fall 4a: Der resultierende DAG enthält die DAG-Knoten D1=(a, y, b, UNKNOWN) und D2=(b, x, fc, ns), wobei a die ID von D1 und b die ID von D2 ist. Anwendung der Operation decompressNode auf den Knoten D1 führt somit zu einem Aufruf der Operation decompressNode(D2, DAGEntry.FirstChild). Diese erzeugt die Folge firstChild('x') $\otimes$ vFC $\otimes$ vNS.
- Fall 4b: Der resultierende DAG enthält die DAG-Knoten D1=(a, y, UNKNOWN, b) und D2=(b, x, fc, ns), wobei a die ID von D1 und b die ID von D2 ist. Anwendung der Operation decompressNode auf den Knoten D1 führt somit zu einem Aufruf der Operation decompressNode(D2, DAGEntry.NextSibling). Diese erzeugt die Folge nextSibling('x') $\otimes$ vFC $\otimes$ vNS.

Da die Fälle 1a bis 4b alle vollständigen binären Teilbäume umfassen, gilt somit die Behauptung.

□

5.2 DAG-Kompression für unendlich lange XML-Datenströme

Während der Kompression – also während der Berechnung des DAGs – steigt bei unendlich langen XML-Datenströmen die Größe des DAGs immer weiter an. Dies stellt jedoch kein Problem dar, da man in diesem Fall den DAG beim Erreichen einer vorher festgelegten Größe "abschließen" und zum Empfänger versenden kann. Dies entspricht also der Anwendung eines diskreten Fensters auf dem Eingangsstrom, so dass immer nur innerhalb eines Fensters Verweise auftreten können, jedoch nicht von Fenster zu Fenster.

Da laut [11] die durchschnittliche Tiefe eines XML-Dokuments bei 4 Knoten liegt (wobei 99% aller Dokumente eine Tiefe von maximal 8 Knoten haben, und die maximal erreichte Tiefe 135 war), ist zwar die Tiefe eines unendlichen XML-Datenstroms beschränkt, dies gilt jedoch nicht für die Breite. Daher steigt auch die Größe des Stacks, der alle offenen Knoten, also den Pfad zum aktuellen Knoten bestehend aus first-child- und next-sibling-Achsen, enthält. Da für diese Knoten allerdings noch mindestens eine Information fehlt, können diese Knoten noch nicht zum Empfänger versendet werden.

Beispiel 5.5 *Abbildung 5.3 zeigt schematisch eine mögliche Überlaufsituation. Die Knoten v1 bis v7 sind jeweils noch nicht komplett abgearbeitet, da der next-sibling-Knoten und eventuell der first-child-Knoten noch nicht abgearbeitet*

wurden und daher die entsprechende ID für einen Verweis noch unbekannt ist. Die grauen Vierecke D1 bis D4 stellen in dieser Abbildung einen bereits fertig abgearbeiteten DAG dar, die Knoten innerhalb dieser Vierecke wurden bereits komplett abgearbeitet und dem DAG hinzugefügt.

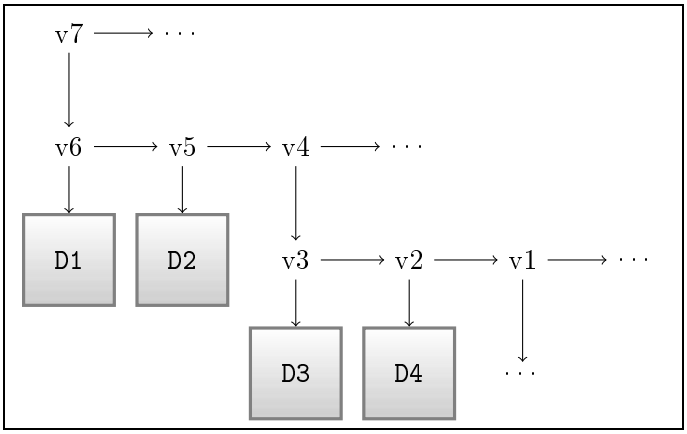

Abbildung 5.3: Zusammenhang zwischen DAG und Stack

Tabelle 5.2 stellt entsprechend den aktuellen Stack dar.

Typ	ID	Label	First-Child	Next-Sibling
NS		v1		
NS		v2	D4	
FC		v3	D3	
NS		v4		
NS		v5	D2	
FC		v6	D1	
FC		v7		

Tabelle 5.2: Stack zum Beispiel

Um nun Teile des Stacks in den DAG verdrängen zu können und somit zum Empfänger zu senden, erhält der oberste auf dem Stack liegende Knoten, der noch keine ID hat (v1), eine ID, so dass die clearStack-Funktion durchgeführt werden kann. Tabelle 5.3 zeigt den Zustand danach, wobei die beiden grau hinterlegten Zeilen nun komplett abgearbeitet wurden und aus dem Stack in den DAG verdrängt wurden.

Dies wird dann mit dem nächsten obersten Stack-Element fortgeführt, das noch keine ID hat. Schließlich verbleiben nur noch die Knoten mit Label v1, v4 und v7, also die Knoten, für die das next-sibling noch unbekannt ist, im Stack.

Typ	ID	Label	First-Child	Next-Sibling
NS	1	v1		
NS	2	v2	D4	1
FC	3	v3	D3	2
NS		v4	3	
NS		v5	D2	
FC		v6	D1	
FC		v7		

Tabelle 5.3: Stack zum Beispiel

Um Teile des Stacks verdrängen zu können, erhält das oberste auf dem Stack befindliche Element TE, welches noch keine ID besitzt, eine ID, und die Funktion clearStack wird durchgeführt, d.h., alle preceding-siblings von TE werden vom Stack verdrängt und in den DAG eingefügt. Dies wird solange wiederholt, bis kein Element ohne ID mehr auf dem Stack liegt. Dadurch verbleiben nur noch diejenigen Knoten auf dem Stack, für die der next-sibling noch unbekannt ist, der Stack enthält also maximal x Elemente, wobei x die Tiefe des aktuellen Knotens innerhalb des XML-Datenstroms ist. Da jedoch – wie bereits oben erwähnt – die Tiefe selbst eines unendlich langen Datenstroms beschränkt ist, ist somit auch die Größe des Stacks beschränkt, daher kann mit Hilfe dieser Überlaufbehandlung das DAG-Verfahren auch auf unendliche XML-Datenströme angewandt werden.

Wenn also DAG und Stack zusammen eine vorher festgelegte Fenstergröße erreichen, so wird – wie oben beschrieben – der Stack bereinigt, und sowohl DAG als auch bereinigter Stack werden an den Empfänger versandt. Die Stack-Knoten unterscheiden sich dadurch von den DAG-Knoten, dass sie keinen Eintrag für den next-sibling-Knoten enthalten. Beim Sender wird der DAG gelöscht, der Stack jedoch für die weitere Kompression weiterverwendet.

Der Dekompressor auf der Empfängerseite kann die Dekompression wie in Kapitel 5.1.5 beschrieben durchführen, wobei der Wurzelknoten, bei dem die Dekompression gestartet wird, das unterste Stack-Element ist. Sobald die Dekompression auf einen leeren Verweis stößt, wurde das Ende des aktuellen Fensters erreicht. In diesem Fall kann auf Empfängerseite der DAG gelöscht werden, und ein neues DAG- und Stack-Paket empfangen werden. Auch der Stack kann auf Empfängerseite gelöscht werden, da alle im bisherigen Stack enthaltenen Elemente entweder im neuen DAG oder im neuen Stack enthalten sind. Nach Erhalt eines neuen Pakets fährt die Dekompression bei demjenigen Verweis desjenigen Elementes fort, bei dem die Dekompression beim letzten Paket beendet wurde. Hierbei ist lediglich zu beachten, dass das Element sich

nun statt auf dem Stack auf dem DAG befinden kann, dies kann jedoch einfach mit Hilfe der ID ermittelt werden.

Wird das Verfahren auf unendliche XML-Datenströme angewandt, so erhält man als Resultat im Allgemeinen nicht den *minimalen* DAG, sondern nur einen etwas weniger stark komprimierten DAG, da es nicht möglich ist, auf sehr weit entfernte Teilbäume zu verweisen. Wurde ein Teilbaum aus dem DAG verdrängt und bereits zum Empfänger versendet, so kann auf diesen Teilbaum nicht mehr verwiesen werden, ein wiederholtes Auftreten muss erneut gespeichert werden.

5.3 Navigation entlang von first-child und next-sibling

Ein DAG-Knoten repräsentiert im Normalfall nicht einen einzelnen Knoten des Ursprungs-Dokumentes, sondern eine Menge von Knoten des Ursprungs-Dokumentes. Um einen Knoten des Ursprungs-Dokumentes eindeutig zu identifizieren, wird eine Liste von Paaren von DAG-Knoten-ID sowie dem Typ (First-Child bzw. NextSibling) der eingehenden Kante in diesen Knoten benötigt. Während die Methoden zur Anfrage-Auswertung für die Succinct-Darstellung lediglich eine Position im Bitstrom zur Identifizierung eines Knotens benötigen, erhalten dieselben Methoden für die DAG-Kompression einen Stack mit Paaren (ID, Knotentyp).

Notation 5.1 *Sei xml ein XML-Baum, SS(xml)=*$(ss_1, \ldots, ss_n)$ *der Struktur-Strom zu xml und BS(xml)=*$(bs_1, \ldots, bs_m)$ *der binäre Struktur-Strom zu xml. Dann sagen wir,* ein Knoten k des XML-Baums entspricht dem binären Event $bs_i = bs(i)$ *nach Definition 3.3 genau dann, wenn* $ss_{i+1} = ss(k)$ *nach Definition 3.1.*

Notation 5.2 *Sei BS=*$(s_1, \ldots, s_n)$ *ein binärer Struktur-Strom und DAG ein minimaler DAG. Sei E ein Element mit entsprechendem firstChild-Event* $sE \in BS$ *(bzw. mit nextSibling-Event* $sE \in BS$*). Sei s ein Stack bestehend aus Paaren (ID, Knotentyp). Dann sagen wir* E korrespondiert mit (DAG, s) *genau dann, wenn decompressNode(dag.getEntry(s.top().ID), DAGEntry.FirstChild) (bzw. decompressNode(dag.getEntry(s.top().ID), DAGEntry.NextSibling)) den vollständigen binären Teilbaum BS'* $\subseteq$ *BS startend mit Event sE erzeugt.*

Hierbei ist anzumerken, dass der Stack lediglich für die Implementierung der parent-Achse benötigt wird. Wird ein Verfahren der Anfrage-Auswertung zugrunde gelegt, das lediglich die Vorwärtsachsen benutzt, also nur auf die atomaren Achsen first-child und next-sibling zurückgreift, so kann statt des Stacks auch lediglich die ID des aktuellen Knotens verwendet werden.

5.3.1 first-child

Um aus einem gegebenen Stack den Stack für den first-child-Knoten zu bilden, muss lediglich im DAG die ID fc des first-childs des aktuell oben auf dem Stack liegenden Knotens ermittelt werden, und das Paar (fc, DAGEntry.FirstChild) oben auf den Stack gelegt werden.

Dies wird noch einmal in Algorithmus 5.4 verdeutlicht:

```
1 public Stack getFirstChild(DAG dag, Stack xmlID){
2     int fc=dag.getEntry(xmlID.top().ID).getFirstChild();
3     xmlID.push(fc, DAGEntry.FirstChild);
4     return xmlID;
5 }
```

Algorithmus 5.4: getFirstChild-Funktion für die DAG-Kompression

Der folgende Satz 5.2 zeigt die Korrektheit von Algorithms 5.4.

Satz 5.2. Sei BS ein binärer Struktur-Strom und E ein Element mit firstChild- oder nextSibling-Event $sE \in BS$. Sei dag ein DAG und s ein Stack, so dass E mit dem Tupel (dag, s) korrespondiert. Sei s':= getFirstChild(dag, s);

Dann gilt: E/first-child korrespondiert mit dem Tupel (dag, s').

Beweis. Laut Definitionen 2.4 und 3.3 entspricht E/first-child einem Event $sFC \in S$ mit der Eigenschaft, dass sFC vom Typ first-child ist, und dass sFC im binären Struktur-Strom direkt hinter sE folgt.

Nach Definition 5.3 erhält also der DAG-Knoten dE mit E $\bowtie$ dE als first-child-Zeiger einen Verweis auf den DAG-Knoten dFC mit E/first-child $\bowtie$ dFC. Wegen s':=getFirstChild(dag, s) gilt somit also s'.top().ID=dFC.ID. Somit folgt die Behauptung.

□

5.3.2 next-sibling

Analog zur Implementierung der getFirstChild-Funktion muss zur Implementierung der getNextSibling-Funktion lediglich im DAG dag die ID ns des next-siblings des aktuell oben auf dem Stack liegenden Knotens ermittelt werden und das Paar (ns, DAGEntry.NextSibling) oben auf den Stack gelegt werden.

Dies wird noch einmal in Algorithmus 5.5 verdeutlicht:

```
1 public Stack getNextSibling(DAG dag, Stack xmlID){
2     int ns=dag.getEntry(xmlID.top().ID).getNextSibling();
3     xmlID.push(ns, DAGEntry.NextSibling);
4     return xmlID;
5 }
```

Algorithmus 5.5: getNextSibling-Funktion für die DAG-Kompression

Der folgende Satz 5.3 belegt die Korrektheit von Algorithms 5.5.

Satz 5.3. Sei BS ein binärer Struktur-Strom und E ein Element mit firstChild- oder nextSibling-Event $sE \in BS$. Sei dag ein DAG und s ein Stack, so dass E mit dem Tupel (dag, s) korrespondiert. Sei s':= getNextSibling(dag, s);

Dann gilt: E/next-sibling korrespondiert mit dem Tupel (dag, s').

Beweis. Laut Definitionen 2.4 und 3.3 entspricht E/first-child einem Event $sNS \in BS$ mit der Eigenschaft, dass sNS vom Typ next-sibling ist, und dass BS'=$(sE \otimes FC \otimes sNS) \in BS$, wobei FC ein – eventuell leerer – vollständiger binärer Teilbaum startend mit einem first-child-Event ist.

Nach Definition 5.3 erhält also der DAG-Knoten dE mit E $\bowtie$ dE nach Abarbeitung von FC als next-sibling-Zeiger einen Verweis auf den DAG-Knoten dNS mit E/first-child $\bowtie$ dNS. Wegen s':=getNextSibling(dag, s) gilt somit also s'.top().ID=dNS.ID. Somit folgt die Behauptung.

□

5.4 Unterstützung der DOM-Schnittstelle

5.4.1 Die parent-Achse

Um die Methode getParent zu implementieren, muss nun zum ersten Mal lesend auf den Stack zugegriffen werden. Um zum parent zu navigieren, muss zunächst zum first-child navigiert werden, es muss also solange zum previous-sibling navigiert werden, bis kein weiterer previous-sibling mehr exisitiert. Über die umgekehrte first-child-Kante erreichen wir schließlich den parent-Knoten.

Auf den Stack bezogen, entfernen wir also zunächst solange das oberste Stack-Element, solange dies den Typ NextSibling beinhaltet. Das nun oberste Element ist das first-child. Wird auch dieses vom Stack entfernt, entspricht das oberste Stack-Element dem gesuchten parent-Element.

Dies wird noch einmal in Algorithmus 5.6 verdeutlicht:

```
1 public Stack getParent(DAG dag, Stack xmlID){
2     (int ID, Type t) = xmlID.pop();
3     do{
4         (ID,t) = xmlID.pop();
5     }while(t==DAGEntry.NextSibling)
6     return xmlID;
7 }
```

Algorithmus 5.6: getParent-Funktion für die DAG-Kompression

5.4.2 Einfügen und Löschen

Bei Update-Operationen auf dem DAG muss man beachten, dass ein DAG-Knoten nicht einen einzelnen, sondern eine Menge von Knoten des Original-Dokuments repräsentiert. Da jedoch im allgemeinen Fall nur ein einzelner Knoten aus dieser Menge von Knoten modifiziert werden soll, muss zunächst einmal dieser Knoten aus der Menge von Knoten extrahiert werden. Dies geschieht durch Verdopplung mit Hilfe der Methode extractNode aus Algorithmus 5.7.

5.4.2.1 Hilfsfunktion extractNode

Die Hilfsfunktion extractNode extrahiert einen durch einen Stack aus Paaren (ID, Knotentyp) identifizierten XML-Knoten aus dem DAG-Knoten, der eine Menge von XML-Knoten repräsentiert. Hierzu wird für jeden Knoten die Anzahl der Zeiger, die auf ihn verweisen, benötigt. Diese kann entweder bei der Kompression ermittelt werden und zusammen mit dem Komprimat übermittelt werden oder aber nachträglich auf Empfängerseite ermittelt werden. Wir gehen davon aus, dass eine Methode getNoOfPointer zur Verfügung steht, die zu einem Knoten die Anzahl an Zeigern, die auf diesen verweisen, zurückgibt.

Um den Knoten zu extrahieren, gehen wir im DAG top-down vor. Dies bedeutet, dass wir im Stack – entgegen der sonst üblichen Richtung – von unten nach oben vorgehen (Zeilen 2-12). Sobald der Stack auf einen DAG-Knoten verweist, der mehr als einen Eingangszeiger hat (Zeile 4), duplizieren wir diesen im DAG (Zeilen 5-7), ersetzen den Zeiger des DAG-Knotens, auf den das im Stack darunterliegende Element verweist (Zeilen 8+9), und ersetzen den Stack-Eintrag durch das Duplikat (Zeile 10). Dies wiederholen wir, bis wir am zu ändernden Knoten MN angekommen sind, also bis der komplette Stack durchlaufen wurde. Schließlich ist der komplette Pfad von der Wurzel zu MN eindeutig, der Knoten kann ohne Seiteneffekte geändert werden.

```
1  public Stack extractNode(DAG dag, Stack xmlID){
2      for(int pos=0; pos< xmlID.size(); pos++){
3          (int ID, NodeType nt) = xmlID.get(pos);
4          if(getNoOfPointer(ID)>1){
5              DAGNode dn = DAG.get(ID).clone();
6              int newID = DAG.getNewID();
7              dn.setID(newID);
8              if(nt==NodeType.FirstChild)xmlID.get(pos-1).
                   setFC(newID);
9              else xmlID.get(pos-1).setNS(newID);
10             xmlID.set(pos, dn);
11         }
12     }
13     return xmlID;
14 }
```

Algorithmus 5.7: Hilfsfunktion extractNode für die DAG-Kompression

Beispiel 5.6 *Nachfolgend werde ich nun das Extrahieren eines Knotens erläutern. Grundlage ist wieder das Beispiel aus Listing 2.3, wobei wir in diesem Fall der Einfachheit halber davon ausgehen, dass die ersten beiden Personen jeweils nur die Kindknoten Name und Postfach haben. In Abbildung 5.4(a) wird der Zustand vor dem Update dargestellt, beide Personen-Elemente verweisen auf dieselbe Kindknoten-Liste. Der zweiten Person soll ein Telefon hinzugefügt werden, welches den next-sibling des Postfach-Knotens darstellt. Daher muss der Postfach-Knoten extrahiert werden.*

Der erste Knoten auf dem Pfad von der DAG-Wurzel zu dem zu modifizierenden Knoten, der mehr als einen Eingangszeiger besitzt, ist der Name-Knoten. Daher wird dieser verdoppelt, und die Zeiger werden entsprechend angepasst. Abbildung 5.4(b) zeigt den Zustand nach Verdopplung des Name-Knotens.

Da noch nicht das oberste Stack-Element ereicht wurde, wird die Extraktion noch ein weiteres Mal wiederholt. Der nun zu betrachtende Knoten mit mehr als einem Eingangszeiger ist der Postfach-Knoten. Auch dieser wird verdoppelt, und die Zeiger werden angepasst. Abbildung 5.4(c) zeigt den Zustand nach Verdopplung des Postfach-Knotens.

Da dieser Knoten das oberste Stack-Element darstellt, ist die Extraktion dieses Knotens abgeschlossen, die eigentliche Update-Operation, also das Einfügen eines Telefon-Knotens inklusive Attribut 'modell' kann durchgeführt werden.

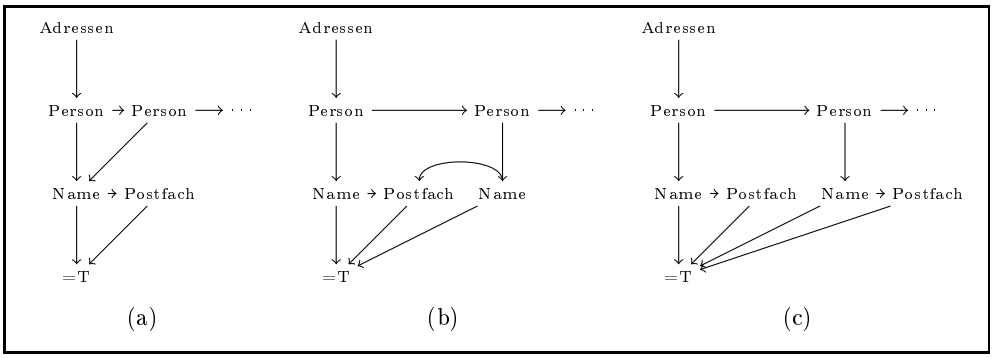

Abbildung 5.4: Extrahieren des Postfachs der zweiten Person

5.4.2.2 Einfügen

Algorithmus 5.8 fügt in einen ursprünglichen DAG old einen neuen DAG new als next-sibling eines durch den Stack node gegebenen Knotens des DAGs old ein. O.B.d.A. gehen wir davon aus, dass beide DAGs unterschiedliche Wertebereiche für Ihre IDs benutzen, so dass keine ID in beiden DAGs verwendet wird. Hierzu muss zunächst einmal der durch den Stack node identifizierte Knoten – wie im vorherigen Abschnitt beschrieben – extrahiert werden (Zeile 2). Danach muss die Wurzel des neuen DAGs new den bisherigen next-sibling nextSib des Knotens node als next-sibling erhalten (Zeilen 3+5). Der Knoten node erhält als neuen next-sibling den Wurzelknoten des neuen DAGs new (Zeile 4). Die Operation old.addAll(DAG new) (Zeile 6) kopiert alle DAG-Knoten des neuen DAGs new in den alten DAG.

```
1 public DAG insert(DAG old, DAG new, Stack node){
2     extractNode(old,node);
3     int nextSib = old.getEntry(node.top().ID).
          getNextSibling();
4     old.getEntry(node.top().ID).setNextSibling(new.
          getRoot());
5     new.getRoot().setNextSibling(nextSib);
6     old.addAll(new);
7     return old;
8 }
```

Algorithmus 5.8: Einfügen für die DAG-Kompression

Eine entsprechende Operation für das Einfügen eines DAGs als first-child eines Knotens kann analog durchgeführt werden, indem in Zeile (3) die Funk-

tion getFirstChild() und in Zeile (4) die Funktion setFirstChild() aufgerufen wird.

Beispiel 5.7 *Betrachten wir wieder das im vorherigen Abschnitt begonnene Beispiel. Nachdem der Knoten extrahiert wurde, soll nun ein DAG bestehend aus Telefon mit Attribut modell und den entsprechenden Text- und Attributwerten eingefügt werden. Abbildung 5.5 zeigt den Zustand des DAGs (beschränkt auf die modifizierte Person und die bisherige Person 3 aus dem Beispiel) nach dem Einfügen. Wie wir sehen können, enthält nun der DAG allerdings eine Redundanz, da durch das Einfügen eine Gleichheit der beiden Teilbäume entstanden ist.*

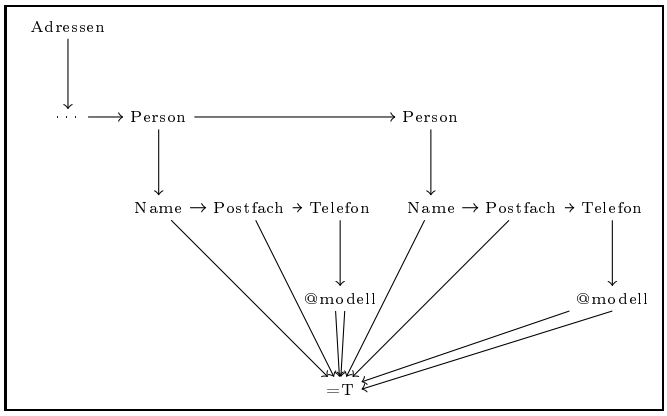

Abbildung 5.5: Einfügen eines neuen Teilbaums

Hätten wir beim Einfügen des neuen DAGs in den alten DAG die Methode DAG.insert benutzt, so dass zumindest der neu eingefügte Teilbaum als bereits vorhanden erkannt worden wäre, hätte dies allerdings zu einem Vorwärtsverweis innerhalb des DAGs geführt, da der sich wiederholende Teilbaum im alten DAG erst nach der Wiederholung durch den neu eingefügten DAG steht. Solche Vorwärtsverweise sind jedoch nicht erwünscht, da dann beispielsweise kein DAG-Event-Strom aus dem DAG generiert werden könnte.

Auch wäre nicht festgestellt worden, dass durch das Neueinfügen ein bereits vorhandener Teilbaum (startend beim Knoten Name) einem anderen bereits vorhandenen Teilbaum gleich geworden ist.

Soll diese Redundanz dennoch vermieden werden, müssten mindestens alle Knoten gleichen Labels auf Redundanzen untersucht werden, was ein Wiedereinfügen vieler DAG-Elemente in den DAG erfordern könnte, und somit einen erheblichen Mehraufwand zur Folge hätte.

Wir erhalten an dieser Stelle also einen Trade-Off zwischen geringerer Kompressionsstärke, aufgrund von Redundanzen durch Update-Operationen und erhöhtem Rechenaufwand bei Update-Operationen.

Es bietet sich allerdings ein Kompromiss an: Update-Operationen werden durchgeführt, ohne eine explizite Redundanz-Vermeidung. Nach einer vorher festgelegten Anzahl x an Update-Operationen wird ein "Wartungsschritt" durchgeführt, also alle DAG-Knoten werden erneut bottom-up in den DAG eingefügt, so dass nach x Update-Operationen wieder der minimale DAG hergestellt wird.

5.4.2.3 Löschen

Algorithmus 5.9 löscht aus dem DAG dag einen durch den Stack node gegebenen Knoten vNode inklusive des Teilbaums, dessen Wurzel er ist. Dies geschieht, indem er den Vorgänger-Knoten von vNode aus dem DAG extrahiert und anschließend den Verweis zum zu löschenden Knoten aus dem DAG entfernt. Zunächst einmal wird ein Stack für den Vorgänger-Knoten erzeugt. Dazu wird das oberste Element aus dem Stack entfernt – der Stack node repräsentiert jetzt nicht mehr den zu löschenden Knoten, sondern dessen Vorgänger-Knoten (Zeile 2). Hierbei werden ID und Knotentyp des zu löschenden Stack-Elements vor der Entfernung dieses Elements gespeichert. Anschließend wird der Vorgänger-Knoten aus dem Stack extrahiert (Zeile 3). Ist der zu löschende Knoten vom Typ next-sibling, so muss der next-sibling-Verweis des previous-siblings auf den next-sibling des zu löschenden Knotens gesetzt werden (Zeilen 4-7). Ist der zu löschende Knoten vom Typ first-child, so muss der first-child-Verweis des parents auf den next-sibling des zu löschenden Knotens gesetzt werden (Zeilen 8-11).

```
1  public DAG remove(DAG dag, Stack node){
2      (int ID, Type t) = node.pop();
3      extractNode(dag, node);
4      if(t==DAGEntry.NextSibling){
5          dag.getEntry(node.top().ID).setNextSibling(
6          dag.getEntry(ID).getNextSibling);
7      }
8      else{
9          dag.getEntry(node.top().ID).setFirstChild(
10         dag.getEntry(ID).getNextSibling);
11     }
12 }
```

Algorithmus 5.9: Löschen für die DAG-Kompression

Die Knoten verbleiben in diesem Fall im DAG. Falls der gelöschte Verweis der einzige Verweis war (es sich bei dem gelöschten Teilbaum also nicht um einen mehrfach verwendeten Teilbaum gehandelt hat), so existiert kein Verweis mehr auf diesen Knoten. Daher empfiehlt es sich auch beim Löschen, neben der im vorherigen Kapitel bereits erwähnten Redundanz durch Updates, von Zeit zu Zeit einen "Wartungsschritt" inklusive Neueinfügen aller DAG-Knoten durchzuführen, um nicht erreichbare Knoten zu löschen.

5.5 Zusammenfassung: Eigenschaften der DAG-Kompression

5.5.1 Kompressionsstärke

Im Gegensatz zur Succinct-Darstellung kann der genaue Speicherbedarf des Komprimats nicht angegeben werden, da dieser sehr stark von der vorhandenen Redundanz innerhalb des Dokumentes abhängt. Ausgehend vom minimalen DAG kann jedoch angegeben werden, wieviel Speicherplatz für jeden DAG-Knoten benötigt wird, die Anzahl DAG-Knoten kann aber nicht in Abhängigkeit der Dokument-Größe angegeben werden.

- Für die Adresse eines DAG-Knotens sowie für die Verweise auf first-child und next-sibling werden je ein Integer-Wert, insgesamt also 3 Integer-Werte benötigt.
- Für das Label eines DAG-Knotens wird ein String benötigt. Da auch hier noch Redundanzen auftreten, die vermeidbar sind, könnte durch den Einsatz einer Symboltabelle hier noch eine stärkere Kompression erreicht werden.

5.5.2 Weitere Eigenschaften

Wie im Verlauf dieses Kapitels gezeigt, hat die DAG-Kompression die folgenden Eigenschaften:

- *Streamingfähig:* Sie ist mit Hilfe der Überlaufkodierung streamingfähig und kann auf unendliche XML-Datenströme angewandt werden, jedoch kann bei unendlichen Datenströmen nicht mehr die Kompressionsstärke des minimalen DAGs erreicht werden, sondern lediglich eine leicht verringerte Kompressionsstärke.
- *Auswertung von Pfad-Anfragen:* Pfad-Anfragen, die mit Hilfe der atomaren Achsen first-child und next-sibling ausgedrückt werden können, können direkt auf dem Komprimat ausgewertet werden. Hierbei kann sehr effizient direkt zum next-sibling navigiert werden.

- *Updates:* Updates können auf dem Komprimat durchgeführt werden, es besteht aber ein Trade-Off: Entweder führen Updates zu einem Verlust der Kompressionsstärke, oder sie erfordern einen Berechnungs-Mehraufwand, der evtl. das Neueinfügen vieler Knoten bedeuten könnte. Dennoch können optimale Updates – also Updates mit Erreichen der optimalen Kompressionsstärke – effizienter durchgeführt werden als die Kombination aus Dekompression, Update auf XML und Kompression.
- *DOM:* Alle Achsen der DOM-Schnittstelle werden voll unterstützt.

6 XML-Kompression durch Eliminierung externer Redundanzen

Das nächste vorgestellte Verfahren – DTD-Subtraktion – entfernt Informationen aus dem Struktur-Anteil eines XML-Dokumentes, die aus der vorhandenen DTD geschlossen werden können.

Beispiel 6.1 *Betrachten wir zur Einführung dieses Verfahrens die DTD aus Listing 2.4 bzw. insbesondere die Elementtyp-Deklaration für das Element 'Person'. Betrachten wir parallel dazu die zweite Person aus dem XML-Dokument in Listing 2.3. Wie wir sehen, schreibt die DTD den ersten Kindknoten – den Knoten 'Name' – fest vor, er muss also – um das Dokument in kompakterer Form darzustellen – nicht gespeichert werden. Als nächstes gibt die Elementtyp-Deklaration durch den Kleene-Operator (*) vor, dass eine Wiederholung bestehend aus entweder 'Strasse' und 'Ort' oder aus 'Postfach' optional gefolgt von dem Element 'Telefon' folgen wird. Hier genügt es, die tatsächliche Anzahl von Wiederholungen – in diesem Fall 2 – zu speichern. Ebenso muss für die binäre Entscheidung, die durch den Oder-Operator (|) dargestellt wird, also, ob 'Strasse' und 'Ort' oder ob 'Postfach' folgt, nur die gewählte Alternative gespeichert werden. Diese binäre Entscheidung kann durch ein Bit im Komprimat repräsentiert werden. Auch der Option-Operator(?) stellt eine binäre Entscheidung dar, nämlich, ob ein 'Telefon'-Element folgt oder nicht, und kann somit durch ein Bit repräsentiert werden. Insgesamt reicht also die Folge int(2), 0, 0, 1, 0 aus, um mit Hilfe der DTD die Kindknoten des zweiten Person-Elements zu repräsentieren. Hierbei steht die int(2) für 2 Wiederholungen, und die 4 Bits (0, 0, 1, 0) stehen für die 4 binären Entscheidungen (zweimal ein Oder-Operator gefolgt von einem Option-Operator).*

Wie an diesem Beispiel zu sehen ist, enthält das XML-Dokument Informationen, die bereits durch die DTD bekannt sind. Haben wir also Kenntnis der DTD, handelt es sich bei diesen Informationen um redundante Informationen, die entfernt werden können, um eine kompaktere Repräsentation des XML-

Dokumentes zu erhalten. Bereits bei diesem kleinen Beispiel enthält die komprimierte Repräsentation 1 Integer-Wert + 4 Bits, während nur die 4 Element-Namen (Name, Strasse, Ort und Postfach) bereits 22 Zeichen benötigen.

6.1 XML-Kompression durch DTD-Subtraktion

Gehen wir davon aus, dass Attribute als besondere Elemente gespeichert werden (mit '@' als Markierungszeichen), ist die DTD eine Menge von Element-Deklarationen. Jede Element-Deklaration enthält den Namen des zu definierenden Elementes – die so genannte *linke Seite* der Element-Deklaration – und einen regulären Ausdruck, der die Liste der Kindknoten definiert – die so genannte *rechte Seite* der Element-Deklaration.

Definition 6.1 (regulärer Ausdruck)**.** Sei Σ die Menge aller Label. Dann gilt:

1. Für $a \in \Sigma$ ist a ein regulärer Ausdruck.
2. EMPTY ist eine regulärer Ausdruck.
3. PCDATA ist eine regulärer Ausdruck.
4. $\emptyset$ ist ein regulärer Ausdruck.
5. R_1, R_2 ist ein regulärer Ausdruck, falls R_1 und R_2 reguläre Ausdrücke sind.
6. $R_1|R_2$ ist ein regulärer Ausdruck, falls R_1 und R_2 reguläre Ausdrücke sind.
7. $R*$ ist ein regulärer Ausdruck, falls R ein regulärer Ausdruck ist.
8. Nur die durch 1.-7. gebildeten Ausdrücke sind reguläre Ausdrücke.

□

Zur Vereinfachung der Darstellung wandeln wir innerhalb der rechten Seite der DTD jeden Teilausdruck der Form 'x+' in die Sequenz 'x,x*' um, wobei x wiederum ein regulärer Ausdruck ist. Weiterhin wandeln wir innerhalb der rechten Seite der DTD jeden Teilausdruck der Form 'x?' in den Ausdruck 'EMPTY|x' um, wobei x wiederum ein regulärer Ausdruck ist. Somit enthalten die rechten Seiten nur noch die Operatoren ',', '|' und '*'.

Definition 6.2. Sei D eine DTD und sei ED ∈ D eine Elementtyp-Deklaration der Form <!ELEMENT name(ED) regExp(ED)>. Sei sb(regExp(ED)) ein Syntaxbaum zum regulären Ausdruck regExp(ED) mit Knotenmenge sb(regExp(ED)).V. Dann bezeichne SB=($\cup_{ED \in D}$ sb(regExp(ED)).V) die *Menge aller Syntaxknoten zu D.*

□

Die dem regulären Ausdruck einer rechten Seite einer DTD-Regel entsprechenden Syntaxbäume enthalten 6 verschiedene Knotentypen:

- *EMPTY*: Dieser Knoten entspricht dem Schlüsselwort *EMPTY* in der DTD. Der Knoten hat im Syntaxbaum keine Kindknoten.
- *PCDATA*: Dieser Knoten entspricht dem Schlüsselwort *PCDATA* in der DTD. Der Knoten hat im Syntaxbaum keine Kindknoten.
- *elem*: Dieser Knoten entspricht einem Nicht-Terminal in der DTD, also einem Element-Label. Der Knoten hat im Syntaxbaum keine Kindknoten. Der Knoten erhält als Parameter *'name'* einen Verweis auf den Wurzelknoten des Syntaxbaums der zum Element-Label gehörenden Definition und als Parameter *'label'* den Bezeichner des Nicht-Terminals.
- *seq*: Dieser Knoten entspricht dem rechts-assoziativen ','-Operator in der DTD. Er hat im Syntaxbaum als linken Kindknoten den ersten Parameter und als rechten Kindknoten den zweiten Parameter dieses Operators.
- *choice*: Dieser Knoten entspricht dem rechts-assoziativen '|'-Operator in der DTD. Er hat im Syntaxbaum als linken Kindknoten den ersten Parameter und als rechten Kindknoten den zweiten Parameter dieses Operators.
- *kleene*: Dieser Knoten entspricht dem '*'-Operator in der DTD. Er hat im Syntaxbaum als linken Kindknoten den Parameter dieses Operators und hat keinen rechten Kindknoten.

Im Nachfolgenden liefert die Funktion x.left für einen Knoten x im Syntaxbaum den linken Kindknoten und die Funktion x.right den rechten Kindknoten.

Beispiel 6.2 *Betrachten wir die folgende DTD-Regel aus Beispiel 2.4:*
<!ELEMENT Adresse (Name, ((Postfach | (Strasse, Ort)), Telefon?))>*
Zunächst einmal wird der ?-Operator ersetzt, so dass wir als rechte Seite der DTD-Regel den folgenden regulären Ausdruck erhalten
(Name, ((Postfach | (Strasse, Ort)), (EMPTY | Telefon)))*
Der entsprechende Syntaxbaum hierzu wird in Abbildung 6.1 dargestellt. In dieser Abbildung hat jeder Knoten zusätzlich eine ID (ID1, ..., ID12), die lediglich für Referenzen in den Erläuterungen benötigt werden.

Für XML gilt die Eigenschaft, dass alle Inhaltsmodelle für XML – und somit auch eine DTD – 1-eindeutig sind, d.h. es genügt, das nächste Event im Struktur-Strom anzuschauen, um z.B. für einen Syntaxknoten vom Typ choice entscheiden zu können, welche Alternative gewählt wurde. Die folgende Funktion zur Berechnung der Menge der Start-Terminal-Symbole gibt für jeden Syntaxknoten diejenigen Element-Label an, die als erstes Element erzeugt werden könnten.

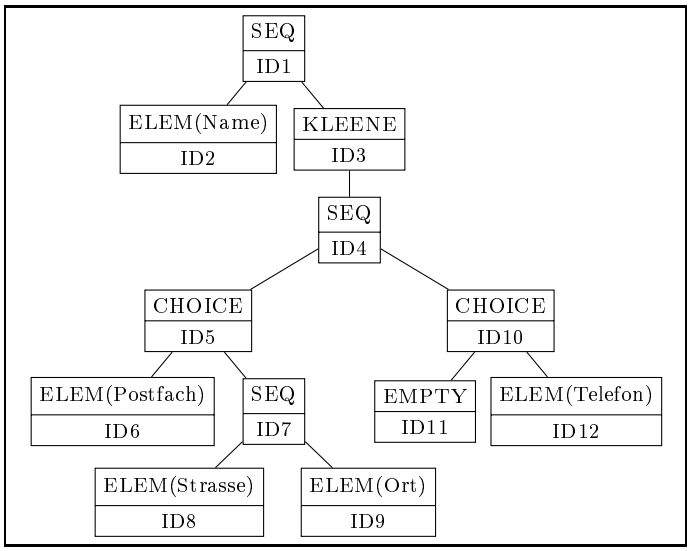

Abbildung 6.1: Schematische Darstellung eines Syntaxbaumes

Definition 6.3 (STS)**.** Sei D eine DTD und sei Σ die Menge aller in D definierten Element-Label. Sei SK die Menge aller Syntaxknoten zu D. Dann ist die Funktion STS: SK $\rightarrow \mathcal{P}(\{EMPTY, PCDATA\} \cup \Sigma)$ definiert durch

$$\text{STS(x):=} \begin{cases} \{x.label\}, & falls\ x\ vom\ Typ\ elem \\ \{PCDATA\}, & falls\ x\ vom\ Typ\ PCDATA \\ \{EMPTY\}, & falls\ x\ vom\ Typ\ EMPTY \\ STS(x.left), & falls\ x\ vom\ Typ\ seq\ und \\ & \neg\,(EMPTY \in STS(x.left)) \\ STS(x.left) \cup STS(x.right), & falls\ x\ vom\ Typ\ seq\ und \\ & EMPTY \in STS(x.left) \\ STS(x.left) \cup STS(x.right), & falls\ x\ vom\ Typ\ choice \\ \{EMPTY\} \cup STS(x.left), & falls\ x\ vom\ Typ\ kleene \end{cases}$$

Die Kompressions- und Dekompressions-Operationen arbeiten auf zwei Listen: dem Struktur-Strom einerseits und dem KST-Strom – dem Komprimat – andererseits. Beide Listen verfügen über die Operationen read, write, skip, getPos und setPos, die wie folgt mit Hilfe eines Arrays implementiert werden können.

```
1 public class
2 List{ Object [] list;
3    int p = 0;    //aktuelle Position
4
```

```
5     public void write(Object o){
6         list[p]=o;
7         p = p+1;
8     }
9
10    public void write(Object o, int pos){
11        list[pos]=o;
12    }
13
14    public Object read(){
15        Object ret = list[p];
16        p = p+1;
17        return ret;
18    }
19
20    public void skip(Object o){
21        if(o==list[p]) p = p+1;
22        else ERROR("Unerwartetes Objekt gefunden");
23    }
24
25    public int getPos(){
26        return p;
27    }
28
29    public void setPos(int pos){
30        p=pos;
31    }
32 }
```

Algorithmus 6.1: Die Klasse List

Die Operation write(Object o) schreibt das übergebene Object o an die aktuelle Position p der Liste und setzt die Position p um 1 weiter. Hierbei ist zu beachten, dass bereits an dieser Stelle stehende Werte überschrieben werden.

Die Operation write(Object o, int pos) schreibt das übergebene Object o an die übergebene Position pos. Die aktuelle Position p der Liste wird hierbei nicht verändert. Werte, die evtl. an Position pos stehen, werden überschrieben.

Die Operation read() gibt das an Position p der Liste stehende Object zurück und setzt die Position p um 1 weiter.

Die Operation skip(Object o) setzt die Position p der Liste um 1 weiter, falls das an aktueller Position p der Liste stehende Object identisch mit dem übergebenen Object ist, andernfalls wird ein Fehler erzeugt.

Die Operation getPos() liefert die aktuelle Position p der Liste zurück und die Operation setPos(int pos) setzt die aktuelle Position p der Liste auf den übergebenen Wert pos.

Da im Folgenden Operationen zur Kompression von XML-Dokumenten definiert werden, die aus einer Folge von Listen-Operationen auf dem Struktur-Strom einerseits und dem Komprimat andererseits bestehen, bezeichnen wir die Operationen write(Object o), write(Object o, int pos), read(), skip(Object o), getPos() und setPos(int pos) als *elementare Listen-Operationen.*

Notation 6.1 *Sei list eine Liste mit den oben definierten elementaren Listen-Operationen write(Object o), write(Object o, int pos), read(), skip(Object o), getPos() und setPos(int pos) und op eine Operation auf list, die aus einer Folge dieser elementaren Listen-Operation besteht. Sei p die Position der Liste list vor Ausführung der Operation op und p' die Position der Liste nach Ausführung von op. Wir sagen OUT ist die Ausgabe von op in list, wenn op keine der Operationen list.skip(Object o) und list.read() enthält, und wenn gilt OUT=(list[p], . . . , list[p']). Analog sagen wir IN ist die Eingabe von op in list, wenn op keine der Operationen list.write(Object o) und list.write(Object o, int pos) enthält, und wenn gilt IN=(list[p], . . . , list[p']).*

Zwei Operationen sind Umkehroperationen, wenn die eine Operation schreibt, was die andere liest und umgekehrt, also wenn die Rolle von Ein- und Ausgabe vertauscht sind, und somit das Ergebnis der Hintereinanderausführung gleich der ursprünglichen Eingabe ist.

Notation 6.2 *Seien op1 und op2 zwei Operationen. Wir sagen, op1 ist eine Umkehroperation von op2, genau dann, wenn die Eingabe von op1 gleich der Ausgabe von op2 und wenn die Ausgabe von op1 gleich der Eingabe von op2 ist.*

Die elementaren Listen-Operationen write(Object o) und read() sind Umkehroperationen zueinander, da write(Object o) gestartet an Position p genau das schreibt, was read() gestartet an Position p liest, also die Ausgabe von write(Object o) der Eingabe von read() entspricht. Da write(Object o) keine Eingabe enthält und read keine Ausgabe, gelten beide Bedingungen, die Operationen sind Umkehroperationen zueinander.

Entsprechend sind auch die elementaren Listen-Operationen write(Object o) und skip(Object o) Umkehroperationen zueinander.

Nun werde ich zunächst die allgemeine Kompressions- und Dekompressions-Operation definieren, die eine Art Weiche für die eigentlichen Kompressions- und Dekompressions-Operationen für die verschiedenen Syntaxknoten-Typen

darstellt. Nachfolgend werde ich für jeden Syntaxknoten-Typ die Grundidee der Kompression, sowie die Operationen comp und decomp definieren.

Schließlich werde ich zeigen, dass comp und decomp Umkehroperationen zueinander darstellen, da die eine schreibt, was die andere liest, und umgekehrt. Für die nachfolgenden Operationen comp und decomp bedeutet dies insbesondere, dass die Operation decomp aus dem Komprimat KST die ursprünglich von der Operation comp gelesene Teilsequenz des Struktur-Stroms S rekonstruiert.

Definition 6.4 (comp, decomp)**.** Sei S ein Struktur-Strom und KST ein Strom aus Integer-Werten, so sind die Operationen *comp(Syntaxknoten n)* und *decomp(Syntaxknoten n)* wie folgt definiert.

```
1  List S;
2  List KST;
3
4  public void comp(Syntaxknoten n)
5  {
6     case (n ist vom Typ)
7     {
8        EMPTY:   compEMPTY();
9        PCDATA:  compPCDATA();
10        elem:    compELEM(n);
11        seq:     compSEQ(n);
12        choice:  compCHOICE(n);
13        kleene:  compKLEENE(n);
14    }
15 }
16 public void decomp(Syntaxknoten n)
17 {
18    case (n ist vom Typ)
19    {
20       EMPTY:   decompEMPTY();
21       PCDATA:  decompPCDATA();
22       elem:    decompELEM(n);
23       seq:     decompSEQ(n);
24       choice:  decompCHOICE(n);
25       kleene:  decompKLEENE(n);
26    }
27 }
```

Algorithmus 6.2: Globale Kompressions- und Dekompressions-Operation

□

6.1.1 EMPTY

Einem Syntaxknoten vom Typ EMPTY entspricht die leere Sequenz im Struktur-Strom. Daher liest die Operation comp weder etwas aus dem Struktur-Strom, noch schreibt sie etwas in das Komprimat.

Definition 6.5 (compEMPTY, decompEMPTY)**.** Sei S ein Struktur-Strom, und KST ein Strom aus Integern, so sind die Operationen *compEMPTY(Syntaxknoten n)* und *decompEMPTY(Syntaxknoten n)* wie folgt definiert.

```
1 List S;
2 List KST;
3
4 public void compEMPTY()
5 {}
6 public void decompEMPTY()
7 {}
```

Algorithmus 6.3: Kompression und Dekompression für EMPTY

□

Lemma 6.1. *decompEMPTY(Syntaxknoten n)* ist die Umkehroperation zu *compEMPTY(Syntaxknoten n)*.

Beweis. Die Behauptung folgt offensichtlich, da bei beiden Operationen weder eine Eingabe noch eine Ausgabe erfolgt. □

6.1.2 PCDATA

Einem Syntaxknoten vom Typ PCDATA entspricht eine Teilfolge <=T>, </=T> im Struktur-Strom. Da der PCDATA-Knoten deterministisch ist, werden das startElement- und das endElement-Event aus dem Struktur-Strom gelesen, es muss jedoch nichts in das Komprimat geschrieben werden.

Definition 6.6 (compPCDATA, decompPCDATA)**.** Sei S ein Struktur-Strom, und KST ein Strom aus Integern, so sind die Operationen *compPCDATA(Syntaxknoten n)* und *decompPCDATA(Syntaxknoten n)* wie folgt definiert.

```
1  List S;
2  List KST;
3
4  public void compPCDATA()
5  {
6      S.skip('<=T>');
7      S.skip('</=T>');
8  }
9  public void decompPCDATA()
10 {
11     S.write('<=T>');
12     S.write('</=T>');
13 }
```

Algorithmus 6.4: Kompression und Dekompression für PCDATA

□

Lemma 6.2. *decompPCDATA(Syntaxknoten n)* ist die Umkehroperation zu *compPCDATA(Syntaxknoten n)*.

Beweis. Bei einem validen Dokument überspringt die Operation *compPCDATA(Syntaxknoten n)* die Folge '<=T></=T>' im Struktur-Strom und schreibt nichts in das Komprimat. Entsprechend liest die Operation *decompPCDATA(Syntaxknoten n)* nichts aus dem Komprimat und schreibt die Folge '<=T> </=T>' in den Struktur-Strom. Also ist *decompPCDATA(Syntaxknoten n)* die Umkehroperation zu *compPCDATA(Syntaxknoten n)*. □

6.1.3 elem

Ein Syntaxknoten vom Typ elem mit Parameter label='a' entspricht einem Knoten n mit Label 'a' im Struktur-Strom S, also einer Teilfolge <a>S', wobei S' eine Teilfolge von S ist. Da der elem-Knoten deterministisch durch die DTD vorgegeben ist, werden das startElement- und das endElement-Event aus dem Struktur-Strom gelesen, es muss jedoch nichts in das Komprimat geschrieben werden.

Anschließend muss die Kompression des Teilstroms S' gestartet werden, in dem die comp-Funktion für den Syntaxbaum der der Elementdeklaration mit Namen 'a' entspricht angestoßen wird. Dies geschieht im Folgenden durch den Aufruf der Mothode comp(n.name) in Zeile 8 im nachfolgenden Algorithmus.

Definition 6.7 (compELEM, decompELEM)**.** Sei S ein Struktur-Strom und KST ein Strom aus Integern, so sind die Operationen *compELEM(Syntaxknoten n)* und *decompELEM(Syntaxknoten n)* wie folgt definiert.

```
1  List S;
2  List KST;
3
4  public void compELEM(Syntaxknoten n)
5  {
6      S.skip('<'+n.label+'>');
7      //Komprimiere Teilbaum unterhalb dieses Elements
8      comp(n.name);
9      S.skip('</'+n.label+'>');
10 }
11 public void decompELEM(Syntaxknoten n)
12 {
13     S.write('<'+n.label+'>');
14     //Dekomprimiere Teilbaum unterhalb dieses Elements
15     decomp(n.name);
16     S.write('</'+n.label+'>');
17 }
```

Algorithmus 6.5: Kompression undDekompression für ELEM

□

Lemma 6.3. Sei decomp(Syntaxknoten n) die Umkehroperation zu comp(Syntaxknoten n). Dann ist *decompELEM(Syntaxknoten n)* die Dekompression zu *compELEM(Syntaxknoten n)*.

Beweis. Sei lab:=n.label. Die Operation *compELEM(Syntaxknoten n)* überspringt zunächst das Event <lab>, führt dann die Operation comp(n.name) aus und überspringt dann </lab>. Die Eingabe besteht aus <lab>, gefolgt von der Eingabe von comp(n.name), gefolgt von </lab>, die Ausgabe besteht aus der Ausgabe von comp(n.name). Entsprechend besteht die Ausgabe der Operation *decompELEM(Syntaxknoten n)* aus <lab>, gefolgt von der Ausgabe von decomp(n.name), gefolgt von </lab>, die Eingabe besteht aus der Eingabe von decomp(n.name). Unter der Annahme, dass comp(n.name) und decomp(n.name) Umkehroperationen sind, dass also die Ausgabe von comp(n.name) der Eingabe von decomp(n.name) entspricht und umgekehrt, folgt auch, dass *compELEM(Syntaxknoten n)* und *decompELEM(Syntaxknoten n)* zueinander Umkehroperationen sind. □

6.1.4 seq

Ein Syntaxknoten vom Typ seq entspricht einer Folge zweier Teilfolgen im Struktur-Strom. Da der seq-Knoten deterministisch ist, muss nichts gelesen werden und nichts in das Komprimat geschrieben werden, es wird lediglich die Kompression der beiden Teilfolgen ausgeführt.

Definition 6.8 (compSEQ, decompSEQ)**.** Sei S ein Struktur-Strom und KST ein Strom aus Integern, so sind die Operationen *compSEQ(Syntaxknoten n)* und *decompSEQ(Syntaxknoten n)* wie folgt definiert.

```
1  List S;
2  List KST;
3
4  public void compSEQ(Syntaxknoten n)
5  {
6     comp(n.left); //Komprimiere linke Teilfolge
7     comp(n.right);//Komprimiere rechte Teilfolge
8  }
9  public void decompSEQ(Syntaxknoten n)
10 {
11    decomp(n.left); //Dekomprimiere linke Teilfolge
12    decomp(n.right);//Dekomprimiere rechte Teilfolge
13 }
```

Algorithmus 6.6: Kompression undDekompression für SEQ

□

Lemma 6.4. Sei decomp(Syntaxknoten n) die Umkehroperation zu comp(Syntaxknoten n). Dann ist *decompSEQ(Syntaxknoten n)* die Umkehroperation zu *compSEQ(Syntaxknoten n)*.

Beweis. Die Eingabe der Operation *compSEQ(Syntaxknoten n)* besteht aus comp(n.left), gefolgt von der Eingabe von comp(n.right). Entsprechend besteht die Ausgabe der Operation *compSEQ(Syntaxknoten n)* aus der Ausgabe von comp(n.left), gefolgt von der Ausgabe von comp(n.right). Analog besteht die Eingabe der Operation *decompSEQ(Syntaxknoten n)* aus decomp(n.left), gefolgt von der Eingabe von decomp(n.right), und die Ausgabe der Operation *decompSEQ(Syntaxknoten n)* besteht aus der Ausgabe von decomp(n.left), gefolgt von der Ausgabe von decomp(n.right). Unter der Annahme, dass comp (Syntaxknoten n) und decomp(Syntaxknoten n) Umkehroperationen sind, dass also die Ausgabe von comp(Syntaxknoten n) der Eingabe von decomp(Syntaxknoten n) entspricht und umgekehrt, folgt, dass *compSEQ(Syntaxknoten n)* und *decompSEQ(Syntaxknoten n)* zueinander Umkehroperationen sind. □

6.1.5 choice

Ein Syntaxknoten vom Typ choice entspricht einer Teilfolge im Struktur-Strom, wobei der Teilbaum entweder dem linken Kindknoten des choice-Knotens oder dem rechten Kindknoten des choice-Knotens entspricht. Die Kompression kodiert die gewählte Alternative binär und führt anschließend die Kompression für den entsprechenden Syntaxknoten aus.

Definition 6.9 (compCHOICE, decompCHOICE)**.** Sei S ein Struktur-Strom, und KST ein Strom aus Integern, so sind die Operationen *compCHOICE(Syntaxknoten n)* und *decompCHOICE(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public void compCHOICE(Syntaxknoten n)
4  {
5      //Lese nächstes Event, ohne den Positionszeiger zu
           ändern
6      int p = S.getPos();
7      Event e = S.read();
8      S.setPos(p);
9      if (e in STS(n)){//Alternative 0
10         KST.write(0);
11         comp(n.left);
12     }else { //Alternative 1
13         KST.write(1);
14         comp(n.right);
15     }
16 }
17 public void decompCHOICE(Syntaxknoten n)
18 {
19     x = KST.read();
20     if(x==0) decomp(n.left);
21     else decomp(n.right);
22
23 }
```

Algorithmus 6.7: Kompression und Dekompressionfür CHOICE

□

Lemma 6.5. Sei decomp(Syntaxknoten n) die Umkehroperation zu comp(Syntaxknoten n). Dann ist *decompCHOICE(Syntaxknoten n)* die Umkehroperation zu *compCHOICE(Syntaxknoten n)*.

Beweis. Alternative 0 (Das folgende Event im Struktur-Strom entspricht dem linken Teilbaum des Syntaxknotens n): Die Ausgabe von *compCHOICE(Syntaxknoten n)* besteht aus 0, gefolgt von der Ausgabe von comp(n.left). Da der Positionszeiger der Liste S nicht verändert wurde, besteht die Eingabe von *compCHOICE(Syntaxknoten n)* aus der Eingabe von comp(n.left). Entsprechend besteht die Eingabe von *decompCHOICE(Syntaxknoten n)* aus 0, gefolgt von der Eingabe von decomp(n.left), die Ausgabe besteht aus der Ausgabe von decomp(n.left).

Alternative 1 (Das folgende Event im Struktur-Strom entspricht dem rechten Teilbaum des Syntaxknotens n): Die Ausgabe von *compCHOICE(Syntaxknoten n)* besteht aus 1, gefolgt von der Ausgabe von comp(n.right). Da der Positionszeiger der Liste S nicht verändert wurde, besteht die Eingabe von *compCHOICE(Syntaxknoten n)* aus der Eingabe von comp(n.right). Entsprechend besteht die Eingabe von *decompCHOICE(Syntaxknoten n)* aus 1, gefolgt von der Eingabe von decomp(n.right), die Ausgabe besteht aus der Ausgabe von decomp(n.right).

In beiden Fällen ist die Eingabe von *compCHOICE(Syntaxknoten n)* gleich der Ausgabe von *decompCHOICE(Syntaxknoten n)* und umgekehrt, also ist *decompCHOICE(Syntaxknoten n)* die Umkehroperation zu *compCHOICE(Syntaxknoten n)*. □

6.1.6 kleene

Ein Syntaxknoten vom Typ kleene entspricht einer Folge von x Teilfolgen im Struktur-Strom, wobei jede der Teilfolgen dem linken Kindknoten des kleene-Knotens entspricht. In diesem Fall wird die Anzahl x an Wiederholungen in das Komprimat geschrieben.

Definition 6.10 (compKLEENE, decompKLEENE)**.** Sei S ein Struktur-Strom, und KST ein Strom aus Integern, so sind die Operationen *compKLEENE(Syntaxknoten n)* und *decompKLEENE(Syntaxknoten n)* wie folgt definiert.

```
1 List KST;
2
3 public void compKLEENE(Syntaxknoten n)
4 {
```

```
5     //Lese nächstes Event, ohne den Positionszeiger zu
          ändern
6     int p = S.getPos();
7     Event e = S.read();
8     S.setPos(p);
9     int pos = KST.getPos();//Merke aktuelle KST-Position
10    KST.write(0);//Schreibe Platzhalter an aktuelle
          Position
11    int i = 0;
12    while(e in STS(n)){
13        i = i + 1;
14        comp(n.left);
15        p = S.getPos();
16        e = S.read();
17        S.setPos(p);
18    }
19    //Schreibe Anzahl i an gemerkte Position pos
20    KST.write(i, pos);
21 }
22 public void decompKLEENE(Syntaxknoten n)
23 {
24    int i = kst.read();
25    for(int x = 0; x<i; x++)
26        decomp(n.left());
27 }
```

Algorithmus 6.8: Kompression und Dekompressionfür KLEENE

□

Lemma 6.6. Sei decomp(Syntaxknoten n) die Umkehroperation zu comp(Syntaxknoten n). Dann ist *decompKLEENE(Syntaxknoten n)* die Umkehroperation zu *compKLEENE(Syntaxknoten n)*.

Beweis. Die Ausgabe von *compKLEENE(Syntaxknoten n)* besteht aus der Anzahl i der durch den Kleene-Operator erzeugten Teilfolgen gefolgt von i mal der Ausgabe von comp(n.left). Zeilen (7)-(9) sowie Zeilen (16)-(18) der Operation *compKLEENE(Syntaxknoten n)* stellen trotz Aufruf der read-Operation keine Eingabe dar, da der Positionszeiger von S nicht verändert wird, die Eingabe von *compKLEENE(Syntaxknoten n)* besteht also nur aus i mal der Eingabe von comp(n.left). Entsprechend besteht die Eingabe von *decompKLEENE(Syntaxknoten n)* aus der Anzahl i der zu erzeugenden Teilfolgen, gefolgt von i mal der Eingabe von decomp(n.left), und die Ausgabe be-

steht aus i mal der Ausgabe von decomp(n.left). Unter der Annahme, dass decomp(Syntaxknoten n) die Umkehroperation von comp(Syntaxknoten n) ist, ist *decompKLEENE(Syntaxknoten n)* die Umkehroperation zu *compKLEENE(Syntaxknoten n)*. □

Satz 6.1. Die Operation decomp nach Definition 6.4 ist die Umkehroperation zur Operation comp nach Definition 6.4.

Beweis. Betrachtet man das XML-Dokument bottom-up, so sind die Blattknoten vom Typ EMPTY oder PCDATA. Für diese beiden Syntaxknoten-Typen gilt die Behauptung nach Lemmata 6.1 und 6.2 ohne weitere Annahmen. Für die weiteren Syntaxknoten-Typen elem, seq, choice und kleene gilt die Behauptung nach Lemmata 6.3, 6.4, 6.5, 6.6 unter der Annahme, dass die Behauptung für die jeweiligen Kindknoten gilt. Da die Behauptung für die Blattknoten bewiesen ist, folgt sie somit induktiv für alle inneren Knoten. □

Beispiel 6.3 *Listing 6.9 zeigt den KST-Strom, der zum Struktur-Strom aus Listing 3.1 generiert wird. Die erste Zeile gibt die Position innerhalb des Komprimats an, die zweite enthält den jeweiligen Wert im Komprimat, und die dritte Zeile gibt an, ob für diesen Wert Speicherbedarf in Form eines Integers (i) oder in Form eines Bits (b) besteht.*

Hierbei wird die 3 an Position 1 durch den äußeren Kleene-Operator erzeugt (3 Personen), die 1 an Position 2 durch den inneren Kleene-Operator(1 Wiederholung von ((Postfach | (Strasse, Ort)), Telefon?)), die 0 an Position 3 durch den choice-Operator (gewählte Alternative: Postfach), die 0 an Position 4 durch den choice-Operator (kein Telefon, Alternative: EMPTY), usw.

Position	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Wert	*3*	*1*	*0*	*0*	*2*	*1*	*0*	*0*	*0*	*1*	*0*	*1*
(b)it/(i)nt	*i*	*i*	*b*	*b*	*i*	*b*	*b*	*b*	*b*	*i*	*b*	*b*

Listing 6.9: DTD-Subtraktion-Kompression des Beispiels

6.2 DTD-Subtraktion für unendlich lange XML-Datenströme

Betrachtet man die verschiedenen Syntaxknoten-Typen und deren Kompressions- und Dekompressions-Operationen, stellt man fest, dass der Großteil der Operationen nur lokal auf dem Struktur-Strom und dem Komprimat arbeitet, es werden lediglich die Operationen read(), write(Object o) sowie skip(Object o) verwendet. Lediglich die Operation compKLEENE(Syntaxknoten n) greift

auf die Operation write(Object o, int pos) zurück, die an der vorgegebenen Position pos im Komprimat, und somit nicht lokal, schreibt. Führen wir diese Operation auf einem potentiell unendlichen Datenstrom aus, würde das bedeuten, dass wir das Komprimat zwischenspeichern müssten und die Weiterleitung der Ausgabe verzögern müssten, bis die endgültige Anzahl der Wiederholungen ermittelt wurde. Da ein unendlicher Datenstrom allerdings theoretisch eine unendliche Anzahl an Wiederholungen enthalten kann, ist dies praktisch nicht durchführbar.

Um das Kompressionsverfahren DTD-Subtraktion so zu erweitern, dass es auch potentiell unendliche Datenströme komprimieren kann, benötigt man eine Überlaufkodierung für Syntaxknoten vom Typ kleene. Diese könnte z.B. so umgesetzt werden, dass man eine festgelegte Fenstergröße hat, innerhalb derer das Komprimat zwischengespeichert wird und innerhalb derer die Ausgabe des Komprimats verzögert wird. Ist der Zwischenspeicher komplett gefüllt, kann jedoch die Ausgabe noch nicht ausgegeben werden, da für mindestens einen kleene-Knoten noch nicht die Anzahl an Wiederholungen ermittelt werden konnte, wird an diese Stelle ein Markierungszeichen geschrieben, gefolgt von der aktuellen Anzahl an Wiederholungen i. Dieses Markierungszeichen gibt an, dass an späterer Stelle die weitere Anzahl an Wiederholungen folgen wird.

Liest die Dekompressions-Operation das Markierungszeichen, weiß sie, dass zunächst i Wiederholungen folgen, und nach Abschluss dieser Wiederholungen die noch fehlende Anzahl an Wiederholungen folgen wird.

Mit Hilfe dieser Überlaufkodierung für Syntaxknoten vom Typ kleene kann das Kompressionsverfahren DTD-Subtraktion auch für potentiell unendliche Datenströme verwendet werden.

6.3 Navigation entlang von first-child und next-sibling

Nun werde ich die Operationen first und next zur Berechnung von first-child und next-sibling auf dem Komprimat KST vorstellen.

Ein Element E im Struktur-Strom wird eindeutig durch das Tupel (KST, n, p) identifiziert, wobei KST ein Komprimat, n ein Syntaxknoten und p eine Position im Komprimat ist:

Notation 6.3 *Sei $S=(s_1, \ldots, s_n)$ ein Struktur-Strom und KST ein Komprimat. Sei E ein Element mit startElement-Event $sE \in S$ und endElement-Event $eE \in S$. Sei n ein Syntaxknoten und p eine Position. Dann sagen wir* E korrespondiert mit (KST, n, p) *genau dann, wenn die Hintereinanderausführung von KST.setPos(p), decomp(n) die Ausgabe $S':=(sE, \ldots, eE) \subseteq S$ erzeugt.*

6.3.1 first-child

Zu einem Knoten E1 mit startElement-Event sE1 ist entsprechend Definition 2.4 das first-child derjenige Knoten E2, dessen startElement-Event sE2 direkt nach sE1 im SAX-Event-Strom und somit auch im Struktur-Strom folgt. Folgt auf sE1 kein startElement-Event, so hat E1 kein first-child.

Entsprechend simuliert die Operation first die XML Operation first-child auf dem Komprimat KST. Sei E1 ein Element und E2=E1/first-child. Sei weiterhin das Tupel (KST, n, p) das zu E1 korrespondierende Tupel. Dann berechnet die Operation first für die DTD-Subtraktion zu (KST, n, p) die Werte n' und p', mit n'=first(n); p'=KST.getPos(), so dass E2 mit (KST, n', p') korrespondiert. Dies wird in Abbildung 6.2 visualisiert.

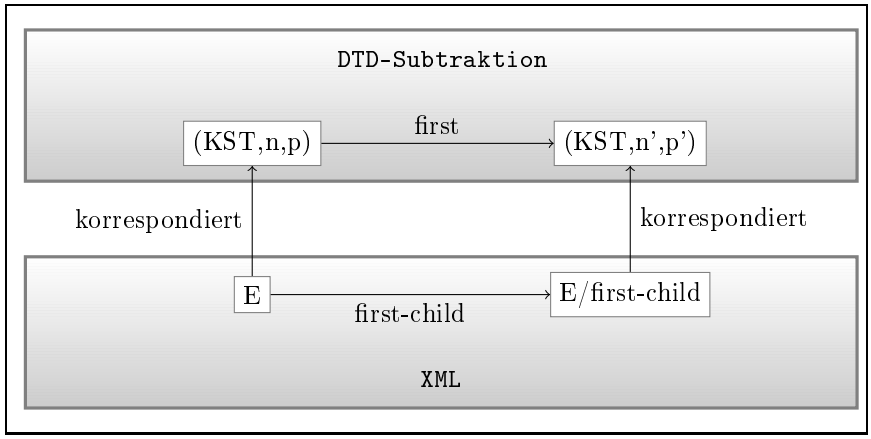

Abbildung 6.2: Schematische Darstellung der first-Operation

Die Operation first simuliert die decomp-Operation, wobei sie im Gegensatz zur decomp-Operation nach Erreichen des ersten PCDATA- oder ELEM-Syntaxknoten abbricht und diesen Knoten ausgibt, so dass die Pointer-Position im Komprimat auf dessen Position verweist. Sie bleibt vor der ersten Ausgabe in den Struktur-Strom stehen, so dass die anschließende Dekompression auf dem KST mit der berechneten aktuellen Position p' diesen Knoten erzeugt.

Ähnlich wie bei der Dekompression existiert eine übergeordnete first-Operation, die eine Weiche für die first-Operationen der unterschiedlichen Knotentypen darstellt.

Definition 6.11 (first)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *first(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten first(Syntaxknoten n)
4  {
5     case (n ist vom Typ)
6     {
7        EMPTY:   return firstEMPTY(n);
8        PCDATA:  return firstPCDATA(n);
9        elem:    return firstELEM(n);
10       seq:     return firstSEQ(n);
11       choice:  return firstCHOICE(n);
12       kleene:  return firstKLEENE(n);
13    }
14 }
```

Algorithmus 6.10: Die globale first-Operation

□

Der EMPTY-Knoten erzeugt keine Ausgabe und hat auch keine weiteren Kindknoten. Die first-Operation des EMPTY-Knotens ändert den Zustand nicht und gibt null zurück.

Definition 6.12 (firstEmpty)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *firstEMPTY(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten firstEMPTY(Syntaxknoten n)
4  {
5     n.mark();
6     return null;
7  }
```

Algorithmus 6.11: Operation first für EMPTY

□

Hinweis: Die Markierung, die durch den Aufruf von n.mark() gesetzt wird, wird im nachfolgenden Abschnitt über das Ermitteln des next-siblings erläutert, da erst dann lesend darauf zugegriffen wird.

Der PCDATA- und der elem-Knoten erzeugen beide eine Ausgabe im Struktur-Strom, ohne dass im Komprimat gelesen werden muss. Daher geben beide sich selbst als Rückgabewert zurück und ändern den Zustand nicht.

Definition 6.13 (firstPCDATA, firstELEM)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so sind die Operationen *firstPCDATA(Syntaxknoten n)* und *firstELEM(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten firstPCDATA(Syntaxknoten n)
4  {
5      n.mark();
6      return n;
7  }
8  public Syntaxknoten firstELEM(Syntaxknoten n)
9  {
10     n.mark();
11     return n;
12 }
```

Algorithmus 6.12: Operation first für PCDATA undELEM

□

Der seq-Knoten erzeugt eine Folge von zwei Teilfolgen, wobei die Teilfolgen auch leer sein können. Er gibt den first-Knoten der ersten Teilfolge aus, falls dieser ungleich null ist, andernfalls gibt er den first-Knoten der zweiten Teilfolge aus. Während er den first-Knoten der ersten und evtl. auch der zweiten Teilfolge ermittelt, wird die aktuelle Pointer-Position p im Komprimat im Allgemeinen verändert.

Definition 6.14 (firstSEQ)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *firstSEQ(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten firstSEQ(Syntaxknoten n)
4  {
5      Syntaxknoten first = first(n.left);
6      if(first!=null) return first;
```

```
7     else return first(n.right);
8  }
```

Algorithmus 6.13: Operation first für SEQ

□

Der choice-Knoten erzeugt eine von zwei möglichen Teilfolgen, wobei im Komprimat die gewählte Alternative kodiert ist. Die first-Operation für den choice-Knoten ermittelt, welche von beiden Alternativen gültig ist, und liefert den first-Knoten der gültigen Alternative zurück. Da die first-Operation hierfür im Komprimat lesen muss, wird hierbei auf jeden Fall die aktuelle Pointer-Position p verändert.

Definition 6.15 (firstCHOICE)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *firstCHOICE(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten firstCHOICE(Syntaxknoten n)
4  {
5     int x = KST.read();
6     if(x==0) return first(n.left);
7     else return first(n.right);
8  }
```

Algorithmus 6.14: Operation first für CHOICE

□

Der kleene-Knoten erzeugt im SAX-Storm eine (evtl. leere) Folge von nichtleeren Teilfolgen von SAX-Events, wobei im Komprimat die Anzahl der Teilfolgen gespeichert ist. Die first-Operation für den kleene-Knoten liefert den first-Knoten der ersten Teilfolge zurück, falls mindestens eine Teilfolge existiert; sonst liefert sie null zurück. Hierbei wird im Komprimat gelesen, somit bleibt die aktuelle Pointer-Position p unverändert.

Definition 6.16 (firstKLEENE)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *firstKLEENE(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten firstKLEENE(Syntaxknoten n)
4  {
5      int x = KST.read();
6      n.max = x;
7      n.count=1;
8      if(x>0) return first(n.left);
9      else return null;
10 }
```

Algorithmus 6.15: Operation first für KLEENE

□

Hinweis: Ähnlich wie der Aufruf von n.mark(), handelt es sich bei den Zuweisungen an n.max und n.count um Markierungen, die erst bei der Ermittlung des next-siblings gelesen werden, und daher erst im nachfolgenden Abschnitt erläutert werden.

Die bisher vorgestellte Menge von Definitionen der first-Operationen führt uns zu dem in Abbildung 6.2 in Abschnitt 6.3.1 skizzierten Hauptsatz:

Satz 6.2. Sei S ein Struktur-Strom und E ein Element mit startElement-Event $sE \in S$ und endElement-Event $eE \in S$. Sei KST ein Komprimat, p eine Position und n ein Syntaxknoten, so dass E mit dem Tupel (KST, n, p) korrespondiert. Seien n'!=null und p' die Rückgabewerte der folgenden Hintereinanderausführung:
KST.setPos(p); n'=first(n); p'=KST.getPos();
Dann gilt: E/first-child korrespondiert mit dem Tupel (KST, n',p')

Beweis. Nach Definition 2.4 gilt für ein Element E mit startElement sE und E/first-child mit startElement sEFC, dass sEFC direkt auf sE im Struktur-Strom folgt. Existiert kein E/first-child, so ist das nächste auf sE folgende Event vom Typ endElement. Entsprechend gilt es im Komprimat zu zeigen, dass die Funktion firs,t angewandt auf einen beliebigen Knoten, den ersten erreichbaren elem- bzw. PCDATA-Syntaxknoten zurückliefert, oder dass sie null zurückliefert, falls kein solcher Syntaxknoten exisitert. Die Blattknoten im Syntaxbaum sind vom Typ EMPTY oder PCDATA bzw. elem. Entsprechend liefert firstEMPTY(Syntaxknoten n) nach Definition 6.12 null zurück, da kein Knoten existiert, und die Operationen firstPCDATA(Syntaxknoten n) und firstELEM(Syntaxknoten n) nach Definition 6.13 liefern den Syntaxknoten n'=n zurück, der Positionszeiger entspricht unverändert der aktuellen Position, also p'=p.

Für jeden inneren Syntaxknoten vom Typ seq bzw. für dessen Operation firstSEQ(Syntaxknoten n) entsprechend Definition 6.14 wird erst überprüft, ob im linken Teilbaum ein elem- bzw. PCDATA-Knoten existiert, in dem für n.left die Operation first(Syntaxknoten n) aufgerufen wird. Liefert diese einen Knoten n' zurück, so ist dieser ein Ergebnis, liefert diese null zurück, wird die Suche im rechten Teilbaum (n.right) fortgeführt. Der Positionszeiger entspricht unverändert der aktuellen Position p'=p+x, wobei x die Anzahl an Positionen ist, die bei der Durchquerung des linken bzw. rechten Teilbaums gelesen wurden.

Für jeden inneren Syntaxknoten vom Typ choice bzw. für dessen Operation firstCHOICE(Syntaxknoten n) entsprechend Definition 6.15 wird die Operation first(Syntaxknoten n) für die gewählte Alternative aufgerufen und deren Ergebnis (null bzw. n') zurückgeliefert. Da die gewählte Alternative gelesen wurde, gilt p'=p+1+x, wobei x die Anzahl an Positionen ist, die bei der Durchquerung der gewählten Alternative gelesen wurden.

Für jeden inneren Syntaxknoten vom Typ kleene bzw. für dessen Operation firstKLEENE(Syntaxknoten n) entsprechend Definition 6.16 wird überprüft, ob mindestens eine Wiederholung vorhanden ist. Da es sich hierbei um eine nicht-leere Teilfolge handeln muss, wird in diesem Fall die Operation first(Syntaxknoten n) für den unter dem kleene-Knoten stehenden Teilbaum n.left aufgerufen und dessen Ergebnis n' zurückgeliefert. Da die gewählte Alternative gelesen wurde, gilt p'=p+1+x, wobei x die Anzahl an Positionen ist, die bei der Durchquerung des linken Teilbaums gelesen wurden.

Die inneren Knoten rufen somit die Operation first(Syntaxknoten n) für die Kindknoten auf bis ein Blattknoten vom Typ PCDATA oder elem erreicht wird und der erste von EMTPY verschiedene Blattknoten als Ergebnis zurückgeliefert werden kann.

Somit korrespondiert das Tupel (KST, n', p') mit dem nächsten zu erzeugenden Knoten, und somit mit dem Knoten E/first-child. □

6.3.2 next-sibling

Zu einem Knoten E1 mit startElement-Event sE1 und endElement-Event eE1 ist entsprechend Definition 2.4 das next-sibling derjenige Knoten E2, dessen startElement-Event sE2 direkt nach dem endElement-Event eE1 im SAX-Event-Strom und somit auch im Struktur-Strom folgt. Folgt auf eE1 kein startElement-Event, hat E1 kein next-sibling.

Dies bedeutet, dass die next-Operation für die DTD-Subtraktion zunächst einmal den aktuellen Teilbaum überspringen muss, um zum „End-Tag“ des aktuellen Knotens zu gelangen. Anschließend muss das als nächstes erzeugte Element ermittelt werden.

Im Folgenden werde ich eine mögliche Implementierung der next-Operation vorstellen, die ein Marker-Konzept zu Hilfe nimmt, um den als nächstes erzeugten Knoten zu ermitteln. Der Marker wird hierbei genutzt, um die bereits besuchten Knoten zu markieren. Die Markierung wird gelöscht, wenn durch einen übergeordneten Kleene-Knoten eine weitere Wiederholung begonnen wird.

Beispiel 6.4 *Betrachten wir zunächst einmal den Syntaxbaum aus Beispiel 6.2 auf Seite 78. Die zweite Person aus Listing 2.3 enthält die Kindknoten Name, Strasse, Ort und Postfach in der angegebenen Reihenfolge. Dementsprechend beginnt der für diesen Teilbaum relevante Ausschnitt aus dem Komprimat in Listing 6.9 an Position 5 und endet bei Position 9. Hierbei steht die 2 an Position 5 für 2 Teilfolgen entsprechend des Teilbaums unter dem Kleene-Knoten, die 1 an Position 6 für Alternative 1 (Strasse, Ort), die 0 an Position 7 für Alternative 0 (EMPTY). Die 0 an Position 8 steht für Alternative 0 (Postfach) gefolgt von einer 0 an Position 9 für Alternative 0 (EMPTY).*

Gesucht sei der next-sibling zum Element E=Ort, welches ein Kindknoten der zweiten Person aus Listing 2.3 ist und welches mit dem Tupel (KST, n, p) korrespondiert, wobei p=7 gilt (da die Alternative Strasse/Ort bereits gelesen wurde) und n der Konten ID9 in Beispiel 6.2 ist. Dies bedeutet, dass Knoten ID9 und alle zuvor besuchten elem-Knoten (Knoten ID2, ID8 und ID9) als besucht markiert sind. Zusätzlich müssen wir im kleene-Knoten speichern, dass es genau 2 Wiederholungen gibt und wir uns aktuell in der 1. Wiederholung befinden.

Um den next-sibling zu ermitteln, müssen wir zunächst den Teilbaum unterhalb von Ort und alle entsprechenden KST-Einträge überspringen. Da Ort in diesem Fall nur PCDATA enthält, existiert kein KST-Eintrag, der zum Teilbaum unterhalb von Ort gehört, die Pointer-Position p im KST bleibt unverändert.

Nun muss der nächste zu erzeugende Knoten ermittelt werden: Da der Knoten mit ID9 als besucht markiert ist, setzen wir die Suche bei dessen parent-Knoten (ID7) fort. Dieser ist vom Typ seq. Da beide Kindknoten bereits als besucht markiert wurden, ist die Verarbeitung dieses Knotens abgeschlossen. Er wird ebenfalls als besucht markiert, und die Suche wird beim nächsten parent-Knoten (ID5) fortgeführt.

ID5 ist vom Typ choice. Ein Knoten vom Typ choice ist abgearbeitet, wenn mindestens einer der beiden Kindknoten (die gewählte Alternative) markiert ist. Da dies der Fall ist, wird auch ID5 als besucht markiert, die Suche fährt bei ID4 fort. In diesem Fall existiert ein noch nicht markierter Kindknoten (ID10), in dem nach dem next-sibling gesucht wird.

ID10 ist vom Typ choice, und es ist noch kein Kindknoten markiert, also wird das nächste Zeichen (0) aus dem KST gelesen, um die gewählte Alternative zu

ermitteln. Da die gewählte Alternative (ID11) EMPTY entspricht, muss die Suche weiter fortgesetzt werden, die Knoten ID11, ID10 und ID 4 werden als besucht markiert.

Der nächste zu betrachtende Knoten ist ID3, ein Knoten vom Typ kleene. Dieser enthält als Markierung die aktuelle Wiederholung (count=1) sowie die maximale Anzahl an Wiederholungen (max=2). Da count<max gilt, existiert noch eine weitere Wiederholung, alle Markierungen im darunterliegenden Teilbaum werden gelöscht, und die Suche wird bei Knoten ID4 und schließlich bei ID5 und ID6 fortgeführt.

Mit ID6 wurde der erste nicht-markierte elem- bzw. PCDATA-Knoten ermittelt, der somit auch das Ergebnis der next-sibling-Suche darstellt. Das Element Postfach bzw. die entsprechenden Einträge KST, Syntaxknoten n'=ID6 und die Position im KST p'=5 wurden ermittelt, so dass das Element Postfach dem Tupel (KST, n', p') entspricht.

Allgemein wird die Suche in dem zum aktuellen Element gehörenden elem-Syntaxknoten, einem Blattknoten, gestartet. Hierbei sind im Syntaxbaum sowohl der aktuelle elem-Syntaxknoten nE1 als auch alle bereits besuchten Syntaxknoten als besucht markiert. Für jeden Syntaxknoten n wird zunächst versucht, den next-sibling Knoten im Komprimat, also den nächsten zu erreichenden elem- bzw PCDATA-Syntaxknoten, unter den descendant-Syntaxknoten zu finden. Wurden alle descendant-Syntaxknoten von n untersucht, jedoch kein elem- bzw. PCDATA-Syntaxknoten gefunden, wird die Suche mit dem parent-Syntaxknoten von n und dessen noch nicht besuchten descendant-Syntaxknoten fortgesetzt. Die Suche kann beendet werden, wenn ein elem- bzw. ein PCDATA-Syntaxknoten erreicht wurde. Wurden alle Syntaxknoten abgearbeitet, ohne dass ein elem- bzw. ein PCDATA-Syntaxknoten erreicht wurde, existiert kein next-sibling-Knoten im Komprimat.

Da das Fortsetzen der Suche beim parent-Knoten immer gleich abläuft, wurde diese Teiloperation in die Operation processParent ausgelagert.

Definition 6.17 (processParent)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *processParent(Syntaxknoten n)* wie folgt definiert.

```
1 List KST;
2
3 public Syntaxknoten processParent(Syntaxknoten n)
4 {
5     n.mark();
```

```
6     if(n.parent==null)return null;
7     else return next(n.parent);
8  }
```

Algorithmus 6.16: Operation processParent

□

Ähnlich wie bei der Dekompression und der first-Operation existiert eine übergeordnete next-Operation, die eine Weiche für die next-Operationen der unterschiedlichen Knotentypen darstellt.

Definition 6.18 (next)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *next(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten next(Syntaxknoten n)
4  {
5     case (n ist vom Typ)
6     {
7        EMPTY:   return nextEMPTY(n);
8        PCDATA:  return nextPCDATA(n);
9        elem:    return nextELEM(n);
10       seq:     return nextSEQ(n);
11       choice:  return nextCHOICE(n);
12       kleene:  return nextKLEENE(n);
13    }
14 }
```

Algorithmus 6.17: Die globale next-Operation

□

Der EMPTY-Knoten stellt keinen Kindknoten und somit auch keinen next-sibling dar. Wird ein EMPTY-Knoten erreicht, wird dieser durch Aufruf der Operation processParent(Syntaxknoten n) als besucht markiert, und die Suche nach dem next-sibling-Knoten beim parent-Knoten fortgeführt.

Definition 6.19 (nextEmpty)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *nextEMPTY(Syntaxknoten n)* wie folgt definiert.

```
1 List KST;
2
3 public Syntaxknoten nextEMPTY(Syntaxknoten n)
4 {
5    return processParent(n);
6 }
```

Algorithmus 6.18: Operation next für EMPTY

□

Der PCDATA-Knoten erzeugt einen Text-Knoten. Da an dieser Stelle nicht zwischen Element-, Attribut- und Text-Knoten unterschieden wird, sondern diese Aufgabe durch das übergeordnete XPath-Framework übernommen wird, wird er gleich wie ein elem-Knoten behandelt.

Da ein Knoten über einen Pfad von first-child- und next-sibling-Achsen erreicht wurde, wurden durch die Aufrufe der Operationen first und next eventuell bereits einige der Knoten markiert. Da sowohl die Operationen firstPCDATA(Syntaxknoten n) und firstELEM(Syntaxknoten n) als auch die Operationen nextPCDATA(Syntaxknoten n) und nextELEM(Syntaxknoten n) einen Aufruf der Methode n.mark() durchführen, falls der Knoten nicht bereits markiert ist[1], ist insbesondere immer der aktuelle PCDATA- bzw. elem-Knoten markiert, für den die Operation next(Syntaxknoten n) initial aufgerufen wurde. Die next-Operation unterscheidet daher zunächst, ob der aktuelle PCDATA- bzw. elem-Knoten markiert ist oder nicht. Ist der Knoten markiert, handelt es sich hierbei um den aktuellen Kontextknoten, für den der next-sibling-Knoten ermittelt werden soll. In dem Fall wird zunächst im Komprimat mit Hilfe der Operation skipSubtree der Teilbaum unter dem elem-Knoten übersprungen und dann wird die Suche beim parent-Knoten fortgesetzt. Ist der Knoten noch unmarkiert, wurde der gesuchte next-sibling-Knoten gefunden, und dieser Knoten wird zurückgegeben.

Die Operation skipSubtree ist hierbei analog zur Operation decomp definiert, nur dass die schreibenden Operationsaufrufe S.write(...) entfallen.

Definition 6.20 (nextPCDATA, nextELEM)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so sind die Operationen *nextPCDATA(Syntaxknoten n)* und *nextELEM(Syntaxknoten n)* wie folgt definiert.

[1] Die Operationen nextPCDATA(Syntaxknoten n) und nextELEM(Syntaxknoten n) führen diesen Aufruf indirekt durch einen Aufruf der Operation processParent(Syntaxknoten n) durch.

```
1  List KST;
2
3  public Syntaxknoten nextPCDATA(Syntaxknoten n)
4  {
5      if(!n.isMarked())return n;
6      else{
7          skipSubtree(n);
8          return processParent(n);
9      }
10 }
11 public Syntaxknoten nextELEM(Syntaxknoten n)
12 {
13     if(!n.isMarked())return n;
14     else{
15         return processParent(n);
16     }
17 }
```

Algorithmus 6.19: Operation next für PCDATA undELEM

□

Der seq-Knoten erzeugt im Struktur-Strom eine Folge von zwei Teilfolgen, wobei die Teilfolgen auch leer sein können. Die next-Operation sucht beginnend mit dem ersten nicht-markierten Knoten x nach dem next-sibling-Knoten, indem sie für x die next-Operation aufruft und deren Ergebnisknoten als Ergebnis weiterreicht. Existiert kein unmarkierter Knoten, wird die Suche beim parent-Knoten fortgesetzt.

Definition 6.21 (nextSEQ)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *nextSEQ(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten nextSEQ(Syntaxknoten n)
4  {
5      if(!left.isMarked()){
6          return next(n.left);
7      }
8      else if(!right.isMarked()){
9          return next(n.right);
```

```
10      }
11      else{
12          return processParent(n);
13      }
14  }
```

Algorithmus 6.20: Operation next für SEQ

□

Der choice-Knoten erzeugt eine von zwei möglichen Teilfolgen, wobei im Komprimat die gewählte Alternative kodiert ist. Ist noch keiner der beiden Kindknoten markiert, wird die Suche bei der im Komprimat kodierten Alternative fortgesetzt. Ist bereits ein Knoten (die gewählte Alternative) markiert, wurde dieser Knoten komplett durchsucht. Die Suche wird deshalb beim parent-Knoten fortgesetzt.

Definition 6.22 (nextCHOICE)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *nextCHOICE(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten nextCHOICE(Syntaxknoten n)
4  {
5      if(left.isMarked() || right.isMarked()){
6          return processParent(n);
7      }
8      else{
9          int x = KST.read();
10         if(x==0) return next(left);
11         else return next(right);
12     }
13 }
```

Algorithmus 6.21: Operation next für CHOICE

□

Der kleene-Knoten erzeugt eine (evtl. leere) Folge von nichtleeren Teilfolgen, wobei im Komprimat die Anzahl der Teilfolgen kodiert ist. Wenn ein kleene-Knoten erreicht wird, bedeutet dies eine neue Wiederholung, und sämtliche Markierungen des Teilbaums unterhalb des kleene-Knotens werden gelöscht. Dies geschieht mit Hilfe der Operation deleteDescendantMarks(). Während für

alle anderen Knotentypen die Markierung lediglich widergibt, ob dieser Knoten bereits besucht wurde oder nicht, enthält die Markierung des kleene-Knotens mehr Informationen: mit Hilfe der Markierung wird sowohl gespeichert, in der wievielten Teilfolge das aktuelle Element enthalten ist (Parameter: count), als auch wieviele Teilfolgen insgesamt vorhanden sind (Parameter: max). Wird die aktuelle Markierung gelöscht, werden die Werte der beiden Parameter max und count auf den Wert 'UNKNOWN' gesetzt, so dass beim nächsten Erreichen des kleene-Knotens der ab dann gültige Wert für max – die Anzahl der Teilfolgen – aus dem Komprimat gelesen werden muss.

Definition 6.23 (nextKLEENE)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *nextKLEENE(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2
3  public Syntaxknoten nextKLEENE(Syntaxknoten n)
4  {
5      if(n.max==UNKNOWN){
6          n.max = KST.read();
7          n.count=0;
8      }
9      if(n.count < n.max){//Es folgt eine weitere
           Wiederholung
10         deleteDescendantMarks();
11         n.count++;
12         return left.next();
13     }
14     else{ //Die letzte Wiederholung wurde abgearbeitet,
           weiter beim Parent
15         return processParent(n);
16     }
17 }
```

Algorithmus 6.22: Operation next für KLEENE

□

Die bisher vorgestellte Menge von Definitionen der next-Operationen führt uns zu dem in Abbildung 6.2 skizzierten Hauptsatz, wobei jeweils first durch next und first-child durch next-sibling ersetzt werden muss:

Satz 6.3. Sei S ein Struktur-Strom und E ein Element mit startElement-Event $sE \in S$ und endElement-Event $eE \in S$. Sei KST ein Komprimat, p

eine Position und n ein Syntaxknoten, so dass E mit dem Tupel (KST, n, p) korrespondiert. Seien n'!=null und p' die Rückgabewerte der folgenden Hintereinanderausführung:
KST.setPos(p); n'=next(n); p'=KST.getPos();
Dann gilt: E/next-sibling korrespondiert mit dem Tupel (KST, n',p').

Beweis. Laut Definition 2.4 ist das next-sibling-Element im Struktur-Strom durch das nächste auf das zugehörige endElement-Event folgende startElement-Event im Struktur-Strom definiert. Entsprechend geht auch die Operation next für die DTD-Subtraktion vor: Zunächst wird in der Operation nextELEM(Syntaxknoten n) nach Definition 6.20 der unterhalb des aktuellen Elements liegende Teilbaum mit Hilfe der Operation skipSubtree() übersprungen, um zum zugehörigen endElement-Event zu gelangen. Bei der Operation nextEMPTY(Syntaxknoten n) für die Blattknoten n vom Typ EMPTY bzw. für markierte elem- bzw. PCDATA-Syntaxknoten wird die Suche beim parent von n fortgeführt (siehe Definitionen 6.19 und 6.20). Für unmarkierte elem- bzw. PCDATA-Syntaxknoten n wird der Knoten n selbst zurückgegeben (siehe Definition 6.20). Für die inneren Syntaxknoten vom Typ seq, choice und kleene wird zunächst für die unmarkierten Kindknoten die Operation next(Syntaxknoten) aufgerufen, um das next-sibling zu ermitteln. Sind alle Kindknoten markiert, wird die Suche beim parent fortgeführt (siehe Definitionen 6.21, 6.22 und 6.23). Wird ein unmarkierter elem- bzw. PCDATA-Knoten n' gefunden, bildet er zusammen mit der aktuellen Position p' im Komprimat KST das mit dem next-sibling-Element im Struktur-Strom korrespondierende Tupel (KST, n', p'). Existiert kein solcher elem- bzw. PCDATA-Knoten, exisitiert auch kein next-sibling. □

6.4 Unterstützung der DOM-Schnittstelle

Um zu zeigen, dass die DTD-Subtraktion mit Hilfe kleiner Modifikationen die komplette DOM-Schnittstelle unterstützt, werde ich in diesem Kapitel eine mögliche Umsetzung der lesenden DOM-Funktion parent sowie der schreibenden DOM-Operationen insert und remove direkt auf dem Komprimat – ohne vorherige Dekompression – beschreiben.

6.4.1 Die parent-Achse

Im Gegensatz z.B. zur Succinct-Darstellung kann die parent-Achse nicht einfach auf dem Komprimat berechnet werden, denn es ist nicht möglich, für einen Knoten E bzw. für dessen korrespondierendes Tupel (KST, n, p) ein Tupel (KST', n', p') zu berechnen, so dass E/parent mit dem Tupel

(KST', n', p') korrespondiert.

Beispiel 6.5 *Betrachten wir das Beispieldokument aus Listing 2.3. Das entsprechende Komprimat wird in Listing 6.9 dargestellt. Gesucht werde das parent zum Postfach der ersten Person. Dieser Postfach-Knoten entspricht dem Tupel (KST, n, p), wobei n der elem(Person)-Knoten (ID6) im Syntaxbaum in Abbildung 6.1 ist, und p=3, die Pointer-Position zeigt auf die erste 0 im KST, die für die gewählte Alternative 0 steht. Laufen wir „rückwärts" im KST, kann das Zeichen an Position 2 für zwei verschiedene Syntaxknoten stehen: erstens kann es für den kleene-Knoten stehen, also aussagen, dass es insgesamt eine wiederholte Teilfolge gibt, andererseits könnte es aber auch Teil einer vorangehenden Wiederholung sein, und somit für choice-Knoten (ID10) und die gewählte Alternative „Telefon" stehen.*

Solch einen Beispielfall kann man nicht nur für Wiederholungen, sondern insbesondere auch für choice-Knoten mit Alternativen, die eine unterschiedliche Anzahl Zeichen im KST bewirken, konstruieren.

Um dennoch eine DOM-Schnittstelle für die DTD-Subtraktion unterstützen zu können, kann man die parent-Achse z.B. mit Hilfe eines Stacks, der alle ancestor-Knoten des aktuell betrachteten Knotens enthält, realisieren.

Da es sich laut [11] bei einer Untersuchung von über 190.000 im Web verfügbaren XML-Dokumenten gezeigt hat, dass die durchschnittliche Tiefe eines XML-Dokuments bei 4 Knoten liegt (wobei 99% aller Dokumente eine Tiefe von maximal 8 Knoten haben, und die maximal erreichte Tiefe 135 war), ist die zu erwartende Größe des Stacks entsprechend begrenzt, so dass kein allzu großer Overhead entsteht.

Hierzu muss jedes Mal, wenn ein Syntaxknoten durch die first-Operation zurückgeliefert wird, dieser zusammen mit der aktuellen Position auf den Stack gelegt werden, und jedes Mal, wenn ein Syntaxknoten durch die next-Operation zurückgeliefert wird, muss dieser zusammen mit der aktuellen Pointer-Position p im KST die oberste Stack-Ebene ersetzen. Um zum parent zurückzukehren, wird das oberste Element des Stacks zurückgeliefert und vom Stack entfernt. Dies führt zu den folgenden Neudefinitionen der first- und der next-Operation und zur folgenden Defintion der parent-Operation:

Definition 6.24 (firstPCDATA, firstELEM)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so sind die Operationen *firstPCDATA(Syntaxknoten n)* und *firstELEM(Syntaxknoten n)* wie folgt definiert.

```
1  Stack ancestor;
2
3  public Syntaxknoten firstPCDATA(Syntaxknoten n)
4  {
5     n.mark();
6     ancestor.push((n, KST.getPos()));
7     return n;
8  }
9  public Syntaxknoten firstELEM(Syntaxknoten n)
10 {
11    n.mark();
12    ancestor.push((n, KST.getPos()));
13    return n;
14 }
```

Algorithmus 6.23: Operation first für PCDATA undELEM

□

Zusätzlich zu der aus Definition 6.13 bekannten Funktionalität werden vor Rückgabe des Syntaxknotens der ermittelte Syntaxknoten und die aktuelle Position auf dem Stack ancestor gespeichert (Zeilen 7 und 13).

Definition 6.25 (nextPCDATA, nextELEM)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so sind die Operationen *nextPCDATA(Syntaxknoten n)* und *nextELEM(Syntaxknoten n)* wie folgt definiert.

```
1  List KST;
2  Stack ancestor;
3
4  public Syntaxknoten nextPCDATA(Syntaxknoten n)
5  {
6     if(!n.isMarked()){
7        ancestor.pop();
8        ancestor.push((n, KST.getPos()));
9        return n;
10    }
11    else{
12       skipSubtree(n);
13       return processParent(n);
14    }
```

```
15 }
16 public Syntaxknoten nextELEM(Syntaxknoten n)
17 {
18     if(!n.isMarked()){
19         ancestor.pop();
20         ancestor.push((n, KST.getPos()));
21         return n;
22     }
23     else{
24         return processParent(n);
25     }
26 }
```

Algorithmus 6.24: Operation next für PCDATA und ELEM

□

Zusätzlich zu der aus Definition 6.20 bekannten Funktionalität wird vor Rückgabe des Syntaxknotens das oberste Element des Stacks ancestor durch den ermittelte Syntaxknoten und die akutelle Position ersetzt (Zeilen 7-8 und 19-20).

Definition 6.26 (parent)**.** Sei KST ein Strom aus Integern mit aktueller Position p = KST.getPos(), so ist die Operation *parent()* wie folgt definiert.

```
1  List KST;
2  Stack ancestor;
3
4  public Syntaxknoten parent()
5  {
6      (Syntaxknoten n, int p) = ancestor.top();
7      ancestor.pop();
8     KST.setPos(p);
9      return n;
10 }
```

Algorithmus 6.25: Operation parent

□

Anstatt Syntaxknoten und Position des parents zu berechnen, werden diese vom obersten Stack-Element ausgelesen, vom Stack verdrängt und zurückgegeben.

6.4.2 Einfügen und Löschen

Die Update-Operationen insert und remove erfordern bei der DTD-Subtraktion im Wesentlichen lokale Änderungen. Bis auf zwei Ausnahmen heisst dies, dass lediglich der komprimierte Teilbaum an der ausgewählten Stelle in das Komprimat geschrieben wird bzw. aus dem Komprimat gelöscht wird. Einzig wenn die Update-Operation die gewählte Alternative oder die Anzahl Wiederholungen einer Teilfolge betrifft, muss zusätzlich die entsprechende übergeordnete Information im Komprimat geändert werden.

Wird eine gewählte Alternative durch eine andere Alternative ersetzt, muss zusätzlich zum Löschen und Neueinfügen der komprimierten Teilbäume das Bit, welches die gewählte Alternative kodiert, geändert werden. Dieses Bit befindet sich direkt vor dem gelöschten Teilbaum und kann beim Ersetzen direkt gelesen werden und mit wenig Aufwand geändert werden.

Wird die Anzahl Wiederholungen eines kleene-Operators geändert, muss der Eintrag an der entsprechenden Stelle im Komprimat erhöht bzw. gesenkt werden. Da jedoch eine Rückwärtssuche der Stelle im Komprimat, wie im vorherigen Abschnitt zur parent-Achse motiviert, nicht möglich ist, müssen in diesem Fall zusätzliche Informationen gespeichert werden, um Updates auf dem Komprimat zu ermöglichen. Zusätzlich zu der maximalen Anzahl an Wiederholungen und der aktuellen Wiederholung, die bereits zur Umsetzung der next-sibling-Achse benötigt wurden, muss zur Laufzeit für jeden bereits besuchten Kleene-Knoten die dazugehörige Position im Komprimat gespeichert werden. Diese kann – entsprechend wie auch die Parameter count und max – beim Durchqueren des Komprimats bei der Ermittlung der Einfüge- bzw. Lösch-Position zur Laufzeit ermittelt werden.

Da es zu jedem Paar aus DTD und XML-Dokument genau ein eindeutiges Komprimat bei der DTD-Subtraktion gibt, gibt es insbesondere für das modifizierte XML-Dokument, in dem die Updates durchgeführt wurden, genau ein eindeutiges Komprimat. Sind die vorgestellten Update-Operationen auf dem Komprimat korrekt, führt also eine Update-Operation U auf dem Komprimat K zu einem Komprimat K', so dass eine anschließende Dekompression von K' zu genau demselben XML-Dokument X' führt, wie die Update-Operation direkt auf dem ursprünglichen XML-Dokument X, so sind daher auch die Update-Operationen optimal. Dies bedeutet, dass Updates auf dem Komprimat zu genau demselben modifizierten Komprimat führen, wie wenn man das Komprimat dekomprimieren würde, Updates auf dem XML-Dokument durchgeführt hätte, und anschließend wieder komprimieren würde.

6.5 Optimierte Darstellung der Kleene-Werte

Bei einer Analyse über verschiedene XML-Test-Dokumente hat sich gezeigt, dass die Integer-Werte, welche die Anzahl Wiederholungen eines Kleene-Operators repräsentieren, über alle Dokumente derselben Häufigkeitsverteilung unterliegen. Diese Beobachtung erlaubt es, eine statische Huffman-Kodierung der kleinsten Werte, konkreter der Werte 0-24 zu berechnen. Werte größer als 24 werden durch ein Markierungstoken gekennzeichnet und anschließend mit Hilfe der Überlaufkodierung für ganzzahlige Integer-Werte kodiert. Dies führt zu einer Kodierung der Kleene-Werte, bei der ein Wert n mit durchschnittlich 0,7*n Bits kodiert wird. Details zu dieser optimierten Darstellung der Kleene-Werte wurden in [16] zur Veröffentlichung eingereicht.

6.6 Zusammenfassung: Eigenschaften der DTD-Subtraktion

6.6.1 Kompressionsstärke

Im Gegensatz zur Succinct-Darstellung kann der genaue Speicherbedarf des Komprimats nicht angegeben werden, da dieser nicht nur vom Dokument, sondern insbesondere auch von der Genauigkeit der DTD abhängt. So kann für ein Dokument mit zwei verschiedenen DTDs die Größe des Komprimats stark variieren: Das Komprimat enthält gar keinen Eintrag, wenn die DTD nur genau diese Dokumentstruktur erlaubt, und es enthält umso mehr Einträge, je öfter *- und |-Operatoren der DTD zur Kompressiohn des XML-Dokumentes benutzt werden. Zusammenfassend lässt sich sagen:

- Für jede Ausprägung eines *-Operators der DTD im Dokument wird 1 Integer im Komprimat benötigt, bzw. unter der Verwendung der statischen Huffman-Kodierung werden durchschnittlich 0,7*n Bits für einen Wert n benötigt.
- Für jede Ausprägung eines |-Operators im Dokument wird 1 Bit im Komprimat benötigt.
- Alle anderen DTD-Operatoren bzw. Syntaxknoten erfordern keinen Speicherplatz im Komprimat.

6.6.2 Weitere Eigenschaften

Wie im Verlaufe dieses Kapitels gezeigt, hat die DTD-Subtraktion noch die folgenden Eigenschaften:

- *Streamingfähig:* Sie ist mit Hilfe der Überlaufkodierung für Syntaxknoten vom Typ kleene streamingfähig.
- *Auswertung von Pfad-Anfragen:* Pfad-Anfragen, die mit Hilfe der atomaren Achsen first-child und next-sibling ausgedrückt werden können, können direkt auf dem Komprimat ausgewertet werden.
- *Updates:* Updates können unbeschränkt auf einem gegebenen Komprimat K durchgeführt werden, insofern, als das Komprimat mit nachfolgender Update-Operation U zu demselben Ergebnis K' führt, wie wenn man erst U zum XML-Dokument X dekomprimiert hätte, die entsprechende Update-Operation U auf X ausgeführt hätte und anschließend wieder zu K"=K' komprimiert hätte. Hierbei wird jedoch zusätzlicher Speicheraufwand benötigt, da für jeden kleene-Operator auf dem Pfad von der Wurzel zum aktuellen Knoten die Stelle seiner Kodierung im Komprimat bekannt sein muss.
- *DOM:* Sie unterstützt indirekt die DOM-Schnittstelle, indem die parent-Achse mit Hilfe eines Stacks implementiert wird.

7 DAG-basierende Kompression

Wie in Kapitel 5 gezeigt verhält sich der DAG bezüglich der Struktureigenschaften wie der XML-Baum, da für jede Kante im XML-Baum eine entsprechende Kante im DAG existiert, und für jeden Knoten im XML-Baum ein entsprechender DAG-Knoten. Daher liegt es nahe, zu versuchen, die beiden anderen vorgestellten Kompressionsverfahren – Succinct-Darstellung und DTD-Subtraktion – so zu verallgemeinern, dass sie statt eines SAX-Event-Stroms auch einen DAG-Event-Strom verarbeiten können.

Hierbei betrachtet man den DAG als einen Baum mit zusätzlichen Rückwärtsverweisen. Sowohl die Succinct-Darstellung als auch die DTD-Subtraktion stellen bereits die Baum-Komponente des DAGs dar, daher diskutiere ich in diesem Kapitel, wie man diese Darstellungen um Rückwärtsverweise erweitern kann.

Diese Kombinationen zweier Kompressions-Ideen sollten jeweils in einer verbesserten Kompression münden, also in einer Kompression mit einem kleineren Komprimat. Um zu verhindern, dass das Einfügen eines Rückwärtsverweises zu einer Aufblähung des ursprünglichen Komprimats führt, werden nur diejenigen Rückwärtsverweise umgesetzt, die "sinnvoll" sind, also zu einer Verkleinerung führen. Genauer gesagt bedeutet dies, dass ein wiederholt vorkommender Teilbaum nur dann durch einen Rückwärtsverweis umgesetzt wird, wenn die Kodierung des Verweises kleiner ist als die Kodierung des wiederholten Teilbaumes. Dies wird insbesondere dazu führen, dass Rückwärtsverweise, die auf sehr kleine Teilbäume verweisen (z.B. mit 1 Knoten) nicht als solche kodiert werden, sondern statt dessen direkt die Kodierung des Teilbaums ins Komprimat geschrieben wird.

7.1 Kodierungsarten für Rückwärtsverweise

In diesem Kapitel diskutiere ich die Vorteile zweier möglicher Implementierungen der Rückwärtsverweise: der Inline-Kodierung einerseits und der Outline-Kodierung andererseits.

Bei der Inline-Kodierung wird ein Rückwärtsverweis direkt an die entsprechende Stelle im Komprimat geschrieben, gekennzeichnet durch ein Markierungstoken. Dadurch muss der Start-Knoten eines Rückwärtsverweises nicht explizit kodiert werden, da er implizit aufgrund der Position bekannt ist.

Bei der Outline-Kodierung werden alle Rückwärtsverweise separat in einem Daten-Strom gespeichert. Dieser muss für jeden Rückwärtsverweis sowohl Start-Knoten als auch Ziel-Knoten explizit speichern.

Im Folgenden werde ich die beiden Kodierungsarten zunächst vorstellen und dann den Speicherverbrauch beider Kodierungsarten vergleichen.

7.1.1 Inline-Kodierung

Bei der Inline-Kodierung wird der Rückwärtsverweis direkt an der entsprechenden Position in den Strom geschrieben. Um einen Verweis zu kennzeichnen, wird eine spezielle Bitfolge – genannt *Markierungstoken* – die solch einen Verweis markiert, vorangestellt. Da natürlich diese Bitfolge auch zufällig im Strom auftreten kann, ohne dass sie einen Verweis darstellt, müssen solche zufälligen Vorkommen entsprechend maskiert werden.

Beispiel 7.1 *Nehmen wir an, das Markierungstoken wäre die Bitfolge 111111. Es muss also gewährleistet sein, dass diese Bitfolge nicht zufällig an anderer Stelle im Strom auftreten kann. Dazu wird jedes Vorkommen der Bitfolge 11111 durch die Bitfolge 111110 ersetzt. So wird zum Beispiel die Folge 111111 durch 1111101 maskiert, und die Folge 111110 durch 1111100.*

Sei allgemein nun M=$(m_1, \ldots, m_n)$, mit $m_i \in \{0,1\}, 1 \leq i \leq n$ das Markierungstoken bestehend aus n Bits. Dann gilt: eine Bitfolge B=$(b_1, \ldots, b_n)$ mit $\forall 1 \leq i < n : b_i = m_i$ wird maskiert durch die Bitfolge B'=$(b_1, \ldots, b_{n-1}, 1 - m_n, b_n)$. Solch eine Maskierung von zufällig auftretenden Markierungstoken kann effizient mit Hilfe eines Automaten umgesetzt werden.

Für XML-Repräsentationen, in denen eine Position p im Strom nicht eindeutig einen Knoten identifiziert – dies ist z.B. bei der DTD-Subtraktion der Fall – muss zusätzlich zu dem Markierungstoken die zur eindeutigen Identifikation noch fehlende Zusatzinformation an Position p geschrieben werden.

Diese Kombination aus Markierungstoken an Position p, eventuell gefolgt von zusätzlichen Identifizierungsinformationen, gibt eindeutig den Start-Knoten des Rückwärtsverweises an. Darauf muss die eindeutige Indentifizierungsinformation für den Ziel-Knoten des Rückwärtsverweises folgen.

7.1.2 Outline-Kodierung

Bei der Outline-Kodierung werden Paare von Identifizierungsinformationen für Start- und Ziel-Knoten eines Rückwärtsverweises in einem extra Strom gespeichert. Dies hat den Vorteil, dass keine Maskierung des bisherigen Stromes erfolgen muss, wodurch kein zusätzlicher Overhead entsteht. Im Vergleich zur Inline-Kodierung muss allerdings zusätzlich die Position des Start-Knotens im Strom gespeichert werden. Ebenfalls erfordert ein zusätzlicher Strom durch Verwaltung und Synchronisation mit den anderen Strömen evtl. einen verwaltungstechnischen Overhead.

7.1.3 Speicherkosten von Inline- und Outline-Kodierung

Betrachten wir zunächst die durch die Inline-Kodierung entstandenen Zusatzkosten. Zunächst einmal müssen im Strom alle zufälligen Auftreten der ersten t-1 Bits des Markierungstoken maskiert werden, wobei t die Länge des Markierungstoken sei. Bei einem Strom der Länge n gibt es also insgesamt (n-(t-1)+1) Bitfolgen der Länge t-1, wobei für jede dieser Bitfolgen die Wahrscheinlichkeit, dass diese Bits gleich den ersten t-1 Bits des Markierungstokens sind, $\frac{1}{2^{t-1}}$ beträgt, wenn wir davon ausgehen, dass alle möglichen Bitfolgen gleichverteilt sind. Nur wenn eine Bitfolge der Länge t-1 den ersten t-1 Bits des Markierungstokens entspricht, erhalten wir 1 Bit Overhead für die Maskierung der Bitfolge. Wir erhalten also als Kosten M für die Markierung in etwa die folgenden Kosten:

$$M(n) \approx (n-(t-1)+1) * \frac{1}{2^{t-1}} Bits$$

Zusätzlich benötigen wir die Kosten zur Speicherung jedes Verweises. Solch ein Verweis besteht aus dem Markierungstoken mit t Bits, evtl. einer zusätzlichen Identifizierungsinformation mit i Bits sowie der Kodierung des Zielknotens mit z Bits. Nehmen wir an, wir haben v Verweise, so ergibt sich für die Gesamt-Verweiskosten V:

$$V(v) = v * (t+i+z) Bits$$

Bei einer Analyse verschiedener DAG-Kompressionen von XML-Dokumenten hat sich gezeigt, dass v=$\frac{n}{r}$ mir r=150 für die Succinct-Darstellung und r=300 für DTD-Subtraktion ein Mittelwert für die Anzahl der sinnvollen Verweise ist, dass es also im Mittel je 150 Bits bzw. 300 Bits der XML-Repräsentation einen sinnvollen Rückwärtsverweis gibt. Somit erhalten wir:

$$V(n) = \frac{n * (t+i+z)}{r} Bits$$

Insgesamt betragen also die durch die Inline-Kodierung entstanden Zusatzkosten

$$I(n) = M(n) + V(n) = (n - (t-1) + 1) * \frac{1}{2^{t-1}} + \frac{n * (t + i + z)}{r} Bits$$

Betrachten wir nun die durch die Outline-Kodierung entstandenen Zusatzkosten. Je Verweis erhalten wir für die Outline-Kodierung zwei Knoten-Kodierungen der Länge z, da wir davon ausgehen können, dass die Identifizierung des Start-Knotens dieselbe Anzahl an Bits benötigt wie die Identifizierung des Ziel-Knotens. Wir erhalten also Zusatzkosten für die Outline-Kodierung in Höhe von:

$$O(n) = v * 2 * z = \frac{2 * n * z}{r} Bits$$

Um nun für die beiden Kodierungsmöglichkeiten herauszufinden, ob die Inline- oder die Outline-Kodierung weniger Zusatzkosten verspricht, muss also die Differenz D der beiden Zusatzkosten gebildet werden.

$$D(n) = I(n) - O(n)$$

$$= (n - (t-1) + 1) * \frac{1}{2^{t-1}} + \frac{n * (t + i + z)}{r} - \frac{2 * n * z}{r} Bits$$

$$= n * [\frac{1}{2^{t-1}} + \frac{t + i - z}{r}] - \frac{t-2}{2^{t-1}} Bits$$

In Abhängigkeit von der Stromgröße n erhalten wir also eine Gerade mit Steigung $m(t) = \frac{1}{2^{t-1}} + \frac{t+i-z}{r}$, für die gilt, dass je geringer die Steigung ist, desto besser ist die Inline-Kodierung im Vergleich zur Outline-Kodierung. Es gilt

$$m(t) < 0 \Leftrightarrow z > \frac{r}{2^{t-1}} + t + i$$

also ist die Inline-Kodierung im Vergleich zur Outline-Kodierung besser, solange die Kosten der Ziel-Kodierung z(t,i) oberhalb der durch t und i vorgegebenen Schranke bleiben.

Um die für die Inline-Kodierung im Vergleich zur Outline-Kodierung günstigste Länge t des Markierungstokens zu ermitteln, müssen wir das Minimum der Steigung m_I von I(n) in Abhängigkeit von t berechnen.

$$m_I(t) = \frac{1}{2^{t-1}} + \frac{t + i + z}{r}$$

$$m_I'(t) = -\frac{ln(2)}{2^{t-1}} + \frac{1}{r}$$

7.2 Succinct-Verfahren mit DAG-Pointern

Bei der Succinct-Darstellung genügt – wie in Kapitel 4 gezeigt – die Position innerhalb des Bitstroms zur eindeutigen Identifizierung eines Knotens. Daher beträgt die Länge i für – in diesem Fall nicht vorhandene – Zusatzinformationen zur Identifizierung des Start-Knotens i=0.

Mit dem für die Succinct-Darstellung sinnvollen Wert r=150 ergibt sich als Nullstelle von $m'_I(t)$ und somit als minimale Steigung t=7,7. Da t jedoch nur ganzzahlige Werte annehmen kann, ist für jede XML-Repräsentation $t \in \{7, 8\}$ so zu wählen, dass $m_I(t)$ minimal ist.

Berechnen wir also die Schranke für die Kosten der Ziel-Kodierung z für die beiden sinnvollen Werte t=7 und t=8 sowie i=0, so erhalten wir z(7,0)>9,34 und z(8,0)>9,17. Dies bedeutet, dass bis zu einer Ziel-Kodierung mit (einschließlich) 9 Bits – entsprechend einer Paketgröße mit bis zu 512 Bits pro Paket – die Outline-Kodierung besser ist als die Inline-Kodierung, aber ab einer Ziel-Kodierung mit 10 Bits ist die Inline-Kodierung besser als die Outline-Kodierung.

Im Folgenden wählen wir nun z=9 Bits für die Kosten der Ziel-Kodierung, um die für die Inline-Kodierung günstigste Tokenlänge t zu berechnen, da mit z=9 eine Paketgröße von bis zu 512 Bits adressiert werden kann. Um zu ermitteln, welche Tokenlänge t für die Inline-Kodierung optimal wäre, müssen wir $m_I(7)$=0,14 und $m_I(8)$=0,12 berechnen. Da bei einer Tokenlänge von t=8 die Steigung der Geraden am geringsten ist, sollte also bei einem Vergleich der beiden Kodierungen t=8 gewählt werden.

Da bei t=8 die Steigung m(8)=0,001 positiv ist, gilt, dass mit steigender Stromgröße n die Outline-Kodierung immer besser wird als die Inline-Kodierung. Um diese Aussage aber richtig bewerten zu können, müssen wir noch den Schnittpunkt mit der x-Achse bzw. die Nullstelle von D(n) berechnen, um zu sehen, ab welcher Stromgröße n die Outline-Kodierung besser ist als die Inline-Kodierung. Die Nullstelle von D(n) liegt bei n=$\frac{450}{11} \approx 41$, also ist ab einer Stromgröße von 41 Bits die Outline-Kodierung besser als die Inline-Kodierung.

Für die Succinct-Darstellung sollte somit bei Paketgrößen von bis zu 512 die Outline-Kodierung gewählt werden, bei größeren Paketgrößen die Inline-Kodierung.

Dies ergibt also bei der Outline-Kodierung pro Verweis zusätzliche Kosten in Höhe von 2 Integern. Da in der Succinct-Darstellung pro Knoten 2 Bits im Bitstrom und 1 Integer im Symbolstrom bzw. in der invertierten Labelliste gespeichert werden, lohnt die Umsetzung eines Rückwärtsverweises bereits ab einer Teilbaumgröße von 2 Knoten. Lediglich Rückwärtsverweise auf Teilbäu-

me der Länge 1 (also z.B. auf den Text-Platzhalter =T) lohnen nicht in der Umsetzung und werden daher wiederholt.

Die in diesem Abschnitt präsentierten Ideen zur Kombination von DAG-Kompression mit Succinct-Kodierung wurden in [15] veröffentlicht.

7.3 DTD-Subtraktion mit DAG-Pointern

Im Gegensatz zur Succinct-Darstellung sind die Informationen zur eindeutigen Identifizierung eines Knotens in der DTD-Subtraktion teurer. Wie in Kapitel 6 gezeigt, benötigen wir zur eindeutigen Identifizierung den Syntaxknoten sowie die Position innerhalb des KST.

Im folgenden beschreibe ich zwei Varianten der Kombination von DTD-Subtraktion mit DAG-Pointern: die naive Variante und eine optimierte Variante, die auf sogenannten expliziten Knoten beruht. In der naiven Variante sind Verweise auf jeden Zielknoten des XML-Dokumentes erlaubt.

Dies bedeutet, dass wir pro Knoten ca. 2 Integer (Syntaxknoten-ID + KST-Position) benötigen, also für einen Verweis in der Outline-Kodierung ca. 4 Integer bzw. 32 Bits. Da im Schnitt ein Knoten mit ca. 2 Bits in der DTD-Subtraktion repräsentiert wird, bedeutet dies, dass erst eine Umsetzung von Rückwärtsverweisen ab einer Teilbaumgröße von mindestens 17 Knoten, also sehr großen Teilbäumen, sinnvoll ist.

Um diese Kosten zu senken, lässt die optimierte Variante die zusätzlich benötigte Identifizierungsinformation – also die Syntaxknoten-ID – aus, indem eindeutig definiert ist, welcher Knoten durch die KST-Position identifiziert wird. Dadurch kann man zwar einige Rückwärtsverweise nicht realisieren, da einige Knoten dadurch nicht adressierbar sind, aber die Menge der sinnvollen Rückwärtsverweise sinkt dadurch nicht, sondern steigt sogar, wie Tests gezeigt haben.

Definition 7.1 (expliziter Knoten)**.** Sei xml ein XML-Dokument, KST das Komprimat zu xml, p eine Position im KST und sei V(p)={x ∈ xml| x korrespondiert mit (KST, n, p), n ist ein Syntaxknoten} die Menge von XML-Knoten, die durch die Position p identifiziert werden. Dann sei der *explizite Knoten* eV(p) zu p definiert durch eV(p)=(y ∈ V(p)| $\forall x \in V(p)$ | y=x ∨ x steht in einem Preorder-Durchlauf von xml vor y).

Diese Definition erlaubt uns, nur Rückwärtsverweise zu realisieren, die einen expliziten Knoten als Start-Knoten haben, da dieser eindeutig aufgrund der KST-Position identifiziert werden kann.

Wir erhalten somit Zusatzkosten für zusätzliche Identifizierungsinformationen in Höhe von i=0. Mit dem für die DTD-Subtraktion sinnvollen Wert r=300 ergibt sich als Nullstelle von $m'_I(t)$ und somit als minimale Steigung

t=8,7. Da t jedoch nur ganzzahlige Werte annehmen kann, ist für jede XML-Repräsentation $t \in \{8, 9\}$ so zu wählen, dass $m_I(t)$ minimal ist. Berechnen wir wieder die Schranke für die Kosten der Ziel-Kodierung z für die beiden sinnvollen Werte t=8 und t=9 sowie i=0, so erhalten wir wieder z(8,0)>10,34 und z(9,0)>10,17. Dies bedeutet, dass bis zu einer Ziel-Kodierung mit (einschließlich) 10 Bits die Outline-Kodierung besser ist als die Inline-Kodierung, aber ab einer Ziel-Kodierung mit 11 Bits ist die Inline-Kodierung besser als die Outline-Kodierung. Da jedoch bei der DTD-Subtraktion die Kodierungskosten für den Zielknoten den Syntax-Knoten sowie die KST-Position umfassen, könnten mit bis zu 10 Bits nur sehr kleine Pakete adressiert werden, so dass der bei der DTD-Subtraktion die Inline-Kodierung gewählt werden sollte.

Wählen wir z.B. z=16 Bits für die Gesamtkosten der Ziel-Kodierung, so erhalten wir $m_I(8)$=0,088 und $m_I(9)$=0,087; es wäre also 9 die optimale Tokenlänge. Durch die Inline-Kodierung erhalten wir nun pro Verweis 9 Bits für das Markierungstoken sowie 16 Bits für die Kodierung des Ziel-Knotens, es lohnen also Rückwärtsverweise von Teilbäumen, deren Kodierung mind. 26 Knoten umfasst. Der erwartete Komprimierungsanstieg durch die Kombination von DAG und DTD-Subtraktion sollte deutlich geringer ausfallen als der erwartete Komprimierungsanstieg durch die Kombination von DAG und Succinct-Darstellung. Dieses wird auch durch dien in Kapitel 11 beschriebenen Messungen bestätigt.

7.3.1 Opimierte Kompression durch Kombination von DAG und DTD-Subtraktion

Um weiterhin die Kosten für Rückwärtsverweise zu senken, kann man versuchen, die Kodierung des Ziel-Knotens effizienter zu gestalten, z.B. indem man auf die Anfragbarkeit verzichtet, und davon ausgeht, dass das Dokument immer komplett dekomprimiert wird.

Ist dies der Fall, so erlauben die Rückwärtsverweise eine einfache Ziel-Kodierung: Da der verwiesene Teilbaum bereits dekomprimiert wurde zum Zeitpunkt des Einlesens des Rückwärtsverweises, kann von vornherein bei der Dekompression die Position jedes erzeugten Knotens innerhalb des SAX-Event-Stroms rekonstruiert werden. Da nun jeder Knoten eine eindeutige Identifizierung hat, nämlich seine Position innerhalb des DAG-Event-Stroms, kann man – wie auch im DAG-Event-Strom – als Kodierung des Zielknotens einfach die "Länge" des Rückwärtsverweises angeben, also den Abstand vom Start-Knoten zum Ziel-Knoten.

Durch solche eine optimierte Ziel-Kodierung – jedoch zum Preis des Verlustes der Anfragbarkeit – werden die Kosten für die Kodierung des Ziel-Knotens auf 8 Bits gesenkt, Umsetzung von Rückwärtsverweisen ab 9 Knoten werden

sinnvoll. Im Gegenzug ist somit aber kein Überspringen von Teilbäumen mehr möglich, zur Anfrage-Auswertung muss die komplette Struktur dekomprimiert werden.

7.4 Dekompression und Navigation

Sowohl Dekompression als auch Navigation können wie in den verwendeten Kompressions-Verfahren – also in diesem Fall wie in den Kapiteln 4 und 6 beschrieben – umgesetzt werden. Lediglich das Fortschreiten im Strom, also das Einlesen des nächsten Strom-Elementes weicht ab: Während bisher einfach das nachfolgende Element des Bitstroms bzw. des KSTs gelesen wurde, müssen bei DAG-basierenden Verfahren auch die Rückwährtsverweise berücksichtigt werden.

Dies kann effizient mit Hilfe eines Stacks zur Verarbeitungszeit geschehen (vergleiche Algorithmus 7.1). Wird der Start-Knoten s eines Rückwärtsverweises erreicht (Zeile 9), so werden die Identifizierungsinformationen von s oben auf dem Stack abgelegt (Zeile 10). Anschließend wird zum Ziel-Knoten des Rückwärtsverweises gesprungen (Zeile 11). Anderenfalls wird einfach die aktuelle Position um 1 erhöht (Zeile 14). Sobald das Ende des aktuellen Teilbaums erreicht wurde (Zeile 6), wird das oberste Stack-Element vom Stack genommen und zum dort gespeicherten (ursprünglichen Start-Knoten) gesprungen (Zeile 7).

```
1  Stack jumpHistory;
2  int currentPos;
3
4  public void moveToNextToken
5  {
6      while(endOfSubtreeReached()){
7          currentPos = jumpHistory.pop() + 1;
8      }
9      if(isPointerSource(currentPos)){
10         jumpHistory.push(currentPos);
11         currentPos = getPointerTarget(currentPos);
12     }
13     else{
14         currentPos++;
15     }
16 }
```

Algorithmus 7.1: moveToNextToken für DAG-basierte Kompression

Mit Hilfe der in Algorithmus 7.1 beschriebenen Methode zum DAG-basierten „Fortschreiten“ im KST bzw. im Bitstrom, können Dekompression und Navigation des Succinct-Verfahrens bzw. der DTD-Subtraktion übernommen werden, lediglich das Berechnen der nächsten Position im KST bzw. im Bitstrom muss mit Hilfe der Funktion moveToNextToken realisiert werden.

8 Integration der Konstanten in das Struktur-Komprimat

In dieser Arbeit habe ich drei Kompressions-Verfahren inklusive zweier Kombinationsmöglichkeiten zur Kompression von XML-Struktur-Strömen dargestellt. Aufgrund der vorangegangenen Trennung von Struktur- und Daten-Strom (siehe Kapitel 3) sind diese Verfahren zur XML-Struktur-Kompression im Prinzip beliebig kombinierbar mit verschiedenen Varianten der Daten-Kompression. Je nach Anwendung und daraus resultierenden Anforderungen können unabhängig voneinander Struktur-Kompressions-Verfahren sowie Daten-Kompressions-Verfahren ausgewählt werden, so dass die Anforderungen möglichst optimal erfüllt werden.

Die in diesem Kapitel vorgestellten Ideen und Ansätze zur Integration von Struktur- und Daten-Strom, sowie zur Kompression des Daten-Stroms sind größtenteils keine eigenen Ideen, sondern stellen im Wesentlichen eine Zusammenfassung von in anderen Kompressoren verwendeten Ideen zur Daten-Kompression dar.

8.1 Zeigerlose vs. verzeigerte Daten-Integration

Vor der Kompression enthalten die voneinander getrennten Ströme Struktur-Strom und Daten-Strom eine implizite, zeigerlose Verbindung: Der Text-Wert zum i-ten Platzhalter im Struktur-Strom befindet sich an der i-ten Position im Daten-Strom.

Eine entsprechende implizite, zeigerlose Daten-Integration kann man auch im Zusammenhang mit den in dieser Arbeit vorgestellten XML-Struktur-Kompressions-Verfahren nutzen: Erhält ein Daten-Kompressions-Verfahren die Ordnung der Text-Werte untereinander, kann also die Konstante, die im ursprünglichen Daten-Strom an Position i stand, eindeutig identifiziert werden, so ist keine weitere Zeiger-Information notwendig, um Dekompression und Anfrage-

Auswertung korrekt auf der Kombination aus XML-Struktur-Kompression und Daten-Kompression durchzuführen. Eine solche implizite, zeigerlose Daten-Integration wird z.B. in [17] verwendet.

Sollen allerdings nur Teile des Komprimats betrachtet werden – z.B. bei partieller Dekompression oder beim Überspringen nicht-relevanter Teilbäume bei der Anfrage-Auswertung – so muss dennoch die zu einem gegebenen Strukturknoten passende Text-Konstante gefunden werden. Dieses kann z.B. dadurch erreicht werden, dass für die 'übersprungenen' Anteile der XML-Repräsentation bekannt ist, wieviele Text-Konstanten in diesen Anteilen enthalten sind. Eine Ermittlung dieser Anzahl zur Laufzeit kommt jedoch prinzipiell einer Dekompression von Teilen der Struktur gleich, und stellt somit einen erheblichen Nachteil dieser zeigerlosen Daten-Integration, insbesondere bei der Anfrage-Auswertung, dar.

Das entgegengesetzte Extrem hierzu wäre eine vollständige Verzeigerung von den Text-Platzhaltern im Struktur-Strom zu den Text-Werten im Daten-Strom (und/oder je nach Anwendung auch umgekehrt). Dies würde bedeuten, dass man z.B. – ähnlich wie die Rückwärtsverweise in Kapitel 7 – entweder das Verweis-Ziel an die Position des Platzhalters in der Struktur-Repräsentation inline kodiert, oder in einer separaten Daten-Struktur outline eine Liste von Verweis-Start (Platzhalter-Position im Struktur-Strom-Komprimat) und Verweis-Ziel (Text-Wert-Position im Daten-Strom-Komprimat) speichert. Eine solche vollständige Verzeigerung wird z.B. in [8] verwendet.

Der Nachteil dieses Verfahrens wird sehr schnell deutlich: Betrachtet man einen binären XML-Baum, so sind etwa die Hälfte aller Knoten Text-Knoten, bei n Knoten erhalten wir $\frac{n}{2}$ Text-Platzhalter. Gehen wir von einer Outline-Kodierung aus, so beinhaltet der Zeiger-Strom pro Verweis mindestens 1 Integer für die Position im Struktur-Strom-Komprimat – vorausgesetzt diese Information reicht zur eindeutigen Identifizierung aus – und 1 Integer für die Position im Daten-Strom-Komprimat. Wir erhalten also einen Overhead von ca. 1 Integer pro Dokument-Knoten.

Um die Vorteile beider Verfahren zu vereinen und die Nachteile zu minimieren, empfiehlt sich der Mittelweg: spärliche Verzeigerung. Hierzu wird für gewisse Knoten (z.B. äquidistant nach jedem 50. Knoten, oder für jeden Wurzelknoten eines Teilbaumes mit einer Tiefe, die ein Vielfaches von 4 ist) in einer Inline- oder Outline-Kodierung gespeichert, wieviele Text-Knoten bis dahin vorhanden sind. Diese Informationen werden mit dem Komprimat gespeichert und übertragen, so dass diese Informationen zur Laufzeit bekannt sind. Wird jetzt zu einem Text-Knoten der konkrete Wert gesucht, so muss zur Ermittlung des Text-Wertes nur entweder rückwärts oder vorwärts bis zu solch einem verzeigerten Knoten navigiert werden, die auf dem Weg liegende Anzahl an Text-Platzhaltern ermittelt werden und auf die dort gespeicherte Anzahl ad-

diert bzw. davon subtrahiert werden. Eine solche spärliche Verzeigerung wird z.B. in [15] verwendet.

Je nach Wahl der Verweis-Dichte überwiegen gewisse Vor- bzw. Nachteile dieser Mischform: Je höher die Dichte gewählt wird, umso stärker nähern wir uns den Vor- und Nachteilen der vollständigen Verzeigerung: Wenig Berechnungs-Overhead bei partieller Dekompression und Anfrage-Auswertung gegenüber gesunkener Kompressionstärke. Entsprechend nähern wir uns umso stärker den Vor- und Nachteilen der zeigerlosen Verzeigerung, je kleiner die Dichte gewählt wird: Optimale Kompressionsstärke gegenüber erhöhtem Berechnungs-Overhead bei partieller Dekompression und Anfrage-Auswertung.

8.2 Kontextlose vs. Kontext-sensitive Daten-Kompression

Der Daten-Strom entsprechend Definition 3.4 enthält nicht nur die eigentlichen Text-Werte, sondern zusätzlich den Element- bzw. Attribut-Namen des übergeordneten Knotens. Diese Information kann als eine Art Kontext-Information betrachtet werden, die zur stärkeren Daten-Kompression verwendet werden kann: Text-Werte, die demselben Element- bzw. Attribut-Namen zugeordnet sind, stammen üblicherweise aus dem selben Wertebereich (das Element 'Postfach' z.B. enthält nur 5-stellige Zahlen, während das Element 'Ort' prinzipiell beliebige Zeichenketten enthält). Fasst man all diese Text-Werte eines Element- bzw. Attribut-Knotens mit identischem Label zusammen in einen Daten-Container und komprimiert diese separat von den anderen Daten-Containern, so erhält man eine stärkere Kompression, als wenn man alle Text-Werte in einem gemeinsamen komprimierten Container speichert [60].

Diese verbesserte Kompression erhält man allerdings wiederum zum Preis eines Overheads in der Zeigerstruktur. Während es bei einem Container für alle Text-Werte gemeinsam genügte, zu speichern, wieviele Text-Werte vor dem aktuellen Text-Wert vorhanden waren, muss man dies nun für alle Container wissen. Dies bedeutet entweder, dass man an einem Verweis-Knoten für jeden Container die Anzahl davorstehender Text-Werte speichern muss, oder man speichert diese Information kontext-bezogen, also z.B. ein Verweis-Knoten mit Name *lab* enthält nur Informationen über den zu *lab* gehörenden Container. Letzere Variante bedeutet entweder eine höhere Verweis-Knoten-Dichte oder ein längeres Navigieren zum nächsten passenden Verweis-Knoten, also einen erhöhten Berechnungs-Aufwand.

Auch hier erhalten wir wieder einen Trade-Off zwischen den zwei möglichen Alternativen kontextloser und kontext-sensitiver Daten-Kompression. Je nach

Anwendung und Anforderungen kann die für diese Anforderungen günstigste Alternative gewählt werden.

8.3 Daten-Kompressions-Verfahren

In diesem Teilkapitel werde ich einige Verfahren zur Daten-Kompression vorstellen sowie deren Eigenschaften diskutieren.

8.3.1 Daten-Liste mit generischem Kompressor

Die kompressionsstärkste Variante, um die in einem Daten-Container enthaltenen Text-Werte zu komprimieren, ist die Kompression des gesamten Containers (im Gegensatz zur Kompression jedes Text-Wertes eines Daten-Containers separat, wie es z.B. bei ALM (siehe Kapitel 8.3.4) der Fall ist) mit Hilfe eines generischen Kompressors. Hierzu eignet sich z.B. das gzip-Verfahren, welches auf Huffman [55] und LZ77 [80] basiert, oder das bzip2-Verfahren, welches unter anderem auf der Burrows-Wheeler-Transformation [27] basiert. Hierbei versucht die Huffman-Kodierung eine möglichst minimale Bit-Darstellung für jedes Zeichen zu finden, und das LZ77-Verfahren ersetzt – ähnlich wie der DAG – wiederholte Teilstrings innerhalb eines Fensters durch einen Verweis. Die Burrows-Wheeler-Transformation stellt eine umkehrbare Umsortierung der Zeichen dar, so dass andere Kompressoren ein besseres Kompressions-Ergebnis erzielen können.

Da die Menge aller Text-Werte eines Containers prozentual eine höhere Redundanz enthält als die einzelnen Text-Werte, erreichen wir durch die Kompression eines kompletten Containers eine erheblich höhere Kompressionsstärke, als wenn diese Verfahren auf die Text-Werte separat angewendet worden wären. Der dadurch erkaufte Nachteil durch die Kompression eines Containers ist, dass beim Zugriff auf eine einzige Konstante der gesamte Container dekomprimiert werden muss. Es müssen also – je nach Verfahren – z.B. alle Text-Werte eines kompletten Fenster-Inhaltes eines unendlichen Daten-Stroms dekomprimiert werden.

Soll das komplette Dokument dekomprimiert werden, ist dies kein Nachteil gegenüber anderen Verfahren – im Gegenteil: die Dekompression des gesamten Containers wird voraussichtlich insgesamt weniger Zeit benötigen als die Dekompression jedes einzelnen Text-Wertes für sich. Betrachten wir aber wieder partielle Dekompression und Anfrage-Auswertung, kann dies – besonders, wenn der Anteil der benötigten Text-Werte gering ist und diese auf viele Container gestreut sind – einen erheblichen Overhead bedeuten.

Im Vergleich der beiden Verfahren erzielt bzip2 die stärkere Kompression, während gzip eine schnellere Laufzeit vorweisen kann – wie auch in Kapitel 11 zu sehen ist.

8.3.2 Huffman

Möchte man den Nachteil umgehen, dass immer komplette Container entpackt werden müssen, bietet sich die Kompression der einzelnen Text-Werte via Huffman-Kodierung [55] an. Die Huffman-Kodierung basiert auf einer Häufigkeitsanalyse der einzelnen Zeichen und generiert eine Zeichenkodierung, die ein Zeichen mit umso weniger Bits kodiert, je häufiger dieses Zeichen insgesamt auftritt. Hierbei wird zwischen statischer und adaptiver Huffman-Kodierung unterschieden. Die adaptive Huffman-Kodierung erfordert ein zweimaliges Durchqueren der zu komprimierenden Daten, um im ersten Schritt die Häufigkeitsanalyse durchzuführen und im zweiten Schritt die Daten entsprechend zu kodieren. Die statische Huffman-Kodierung nutzt eine vorher bekannte Häufigkeitsverteilung (z.B. sprachabhängig), so dass ein einmaliger Durchlauf zum Kodieren ausreicht.

Neben dem Vorteil, dass bei diesem Ansatz nicht mehr der komplette Container dekomprimiert werden muss, hat die Huffman-Kodierung noch einen weiteren Vorteil: gleiche Zeichenketten führen zu gleicher Kodierung. Aufgrund dieser Eigenschaften können Gleichheits-Tests (z.B. bei der Auswertung von Prädikaten) oder auch Präfix-Tests direkt auf dem Komprimat durchgeführt werden und erfordern keinerlei Dekompression.

8.3.3 Sequitur

Sequitur [65] führt in gewissem Maße die Idee von Huffman fort, in dem es nicht nur einzelne Zeichen separat betrachtet, sondern Muster aus mehreren Zeichen.

Sequitur ersetzt sich wiederholende Zeichenfolgen in Zeichenketten mit Hilfe grammatikalischer Regeln. Hierzu fasst es zunächst mehrfach auftretende Digramme – also Paare von Zeichen – zu einer Grammatikregel zusammen, und ersetzt diese Digramme durch einen entsprechenden Regelaufruf. Dies wird hierarchisch fortgesetzt, bis keinerlei Ersetzungen mehr möglich sind. Dabei muss das Dokument nur einmal linear durchquert werden, lediglich die dabei aufgebaute Grammatik muss evtl. mehrfach durchsucht werden bei der Suche nach passenden Regeln. Um eine noch stärkere Kompression zu erreichen, kann im Anschluss an den ersten Durchlauf in einem zweiten Durchlauf für jede Regel eine optimale Regel-ID mit Hilfe des Huffman-Verfahrens anhand der Aufrufhäufigkeit bestimmt werden, um so die Gesamtgröße der Regelaufrufe zu minimieren.

Ebenso wie beim Huffman-Verfahren können Gleichheits- und Präfix-Tests direkt auf dem Komprimat durchgeführt werden und erfordern keinerlei Dekompression.

8.3.4 ALM

ALM (Antoshenkov-Lomet-Murray) [6, 7] ist ein ordnungserhaltendes Kompressions-Verfahren. ALM fasst mehrfach auftretende Teilstrings mit Hilfe eines Wörterbuchs zu kürzeren Token zusammen, wobei die Token $alm(a)$ eines Teilstrings a so gewählt werden, dass gilt: $alm(a) < alm(b) \Rightarrow a < b$ für alle Teilstrings a und b. Dadurch entsteht eine Kompression mit der Eigenschaft, dass Vergleiche mit $=,<,>$ direkt auf dem Komprimat ausgewertet werden können, ohne Dekompression der betroffenen Text-Werte. Dies gilt bei diesem Verfahren jedoch nicht für Präfix-Tests, da ALM es erlaubt, für gleiche Präfixe verschiedene Token zu wählen. Somit kann pro Container ein globales ALM-Wörterbuch erstellt werden, und die einzelnen Text-Werte können mit Hilfe dieses Wörterbuchs dekodiert werden, ohne den gesamten Container zu dekodieren.

ALM erlaubt keine Präfix-Tests auf dem Komprimat, dafür sowohl Gleichheits-Tests als auch Ungleichheits-Tests.

8.4 Fazit: Unabhängige Struktur- und Daten-Kompression

Zusammenfassend kann man sagen, dass die in dieser Arbeit vorgestellten Struktur-Kompressions-Verfahren (Succinct-Verfahren, DAG, DTD-Subtraktion sowie deren Kombinationen) orthogonal und damit beliebig kombinierbar sind mit allen vorgestellten Techniken und Verfahren zur Text-Kompression (z.B. gzip, bzip2, Sequitur, ALM) und mit allen Verfahren zur zeigerlosen oder verzeigerten Daten-Integration sowie der kontextlosen und der kontextsensitiven Daten-Kompression.

9 Effiziente XPath-Auswertung auf XML-Datenströmen

Bisher wurden in dieser Arbeit verschiedene Verfahren zur Struktur-Kompression sowie Kombinations-Möglichkeiten dieser Verfahren vorgestellt und diskutiert, wie diese Verfahren zur Struktur-Kompression mit vorhandenen Verfahren zur Daten-Kompression integriert werden können. Für diese Struktur-Kompressions-Verfahren wurde die Anfrage-Auswertung in Form der Basis-Operationen first-child, next-sibling und parent vorgestellt. Um jedoch herkömmliche XPath-Pfad-Anfragen beantworten zu können, bedarf es noch eines Verfahrens, welches die XPath-Anfragen umwandelt in Aufrufe der Operationen first-child, next-sibling und parent.

In diesem Kapitel werde ich daher zunächst einen Automaten-basierten Ansatz zur Auswertung von XPath-Anfragen auf einem binären SAX-Strom vorstellen. Dieses Verfahren zur Auswertung von XPath-Anfragen auf binären SAX-Event-Strömen wurde in [22] publiziert. Im weiteren Verlauf dieses Kapitels werde ich dann zeigen, wie man das Konzept generalisieren kann, so dass darauf aufbauend ein XPath-Auswerter für beliebige XML-Repräsentationen entwickelt werden kann. Jede XML-Repräsentation muss zur XPath-Auswertung lediglich eine schlanke Schnittstelle bestehend aus den Methoden getFirstChild, getNextSibling, getLabel sowie getType implementieren.

9.1 XPath-Auswertung auf binären SAX-Event-Strömen

Das in diesem Kapitel beschriebene Verfahren zur Auswertung von XPath-Anfragen auf herkömmlichen XML-Strömen basiert auf Automaten zur Repräsentation von XPath-Anfragen, welche einen XML-Strom als Eingabe lesen. Eingaben zu diesem Verfahren sind ein binärer SAX-Event-Strom und eine Anfrage basierend auf den Vorwärtsachsen descendant, descendant-or-self,

self, child und following-sibling. Wie man die übrigen XPath-Achsen auf diese Vorwärtsachsen zurückführen kann, wurde in Kapitel 2.4.1 gezeigt. Ausgaben dieses Verfahrens sind diejenigen Fragmente des SAX-Event-Stroms in Dokument-Reihenfolge, deren Wurzel ein Ergebnis der XPath-Anfrage ist.

Dieses Kapitel gliedert sich wie folgt: Zunächst werden die elementaren XPath-Automaten für die Vorwärtsachsen vorgestellt. Anschließend wird erläutert, wie man für eine XPath-Pfad-Anfrage (ohne Filter) einen XPath-Automaten zusammensetzt und mit dessen Hilfe die Pfad-Anfrage auf dem Eingabe-Strom auswertet. Anschließend wird die Behandlung und Auswertung von Filtern in Anfragen erläutert. Schließlich wird erläutert, wie man aufbauend auf den XPath-Automaten eine Anfrage-Auswertung mit Hilfe der Funktionen getFirstChild, getNextSibling, getLabel und getType umsetzen kann.

9.1.1 Elementare XPath-Automaten

In diesem Kapitel werde ich die elementaren XPath-Automaten vorstellen, die Achsen- und Knotentests eines Location-Steps innerhalb einer Pfad-Anfrage repräsentieren. Entsprechend Definition 2.5 können alle Vorwärtsachsen mit Hilfe der atomaren XPath-Achsen first-child und next-sibling berechnet werden. Aufbauend auf dieser Beobachtung werde ich für alle Vorwärtsachsen einen elementaren XPath-Automaten definieren, der als Eingabe nur die Ereignisse fc (first-child), ns (next-sibling) sowie s (self) gefolgt von einem Knotentest akzeptiert. Aufgrund dieses minimalen Eingabe-Alphabets erhalten wir schlanke Automaten zur XPath-Auswertung, die einerseits ein lineares Durchqueren des Stroms erlauben, andererseits auch sehr speichereffizient darstellbar sind.

Formal ist ein XPath-Automat wie folgt definiert:

Definition 9.1 (XPath-Automat)**.** Ein *XPath-Automat einer XPath-Anfrage path* ist ein nicht-deterministischer endlicher Automat (NFA)

$$XP = (Q, \Sigma, q_0, \delta, f),$$

wobei

- Q die endliche Zustandsmenge ist (wir schreiben hierfür im folgenden auch XP.Q)
- Σ = {fc, ns} ∪ {s::a | a ist ein Element-Name, '@' gefolgt von einem Attribut-Namen, '=' gefolgt von einer Konstante oder $'*'$} ist die Menge der Eingabe-Symbole
- $q_0 \in Q$ ist der Startzustand
- $\delta : Q \times \Sigma \times Q$ ist eine Relation von Übergängen (q_1, e, q_2), wobei q_2 dann ein Nachfolge-Zustand von q_1 ist, wenn das Symbol e vom NFA gelesen wird

- $f \in Q$ ist der Endzustand.

Weiterhin bezeichne $active \subseteq Q$ die Menge der derzeit aktiven Zustände. □

Um den XPath-Automaten einer Pfad-Anfrage zu berechnen, wird zunächst einmal die Pfad-Anfrage in eine Liste von Location-Steps unterteilt und anschließend für jeden Location-Step der elementare XPath-Automat berechnet. Anschließend wird der XPath-Automat zur Anfrage zusammengesetzt.

Ein Location-Step besteht aus einer Achsen-Bedingung sowie aus einem Knoten-Test. Dementsprechend kann jeder Location-Step der Form Achse::Knotentest durch die Folge von Location-Steps Achse::*/self::Knotentest dargestellt werden. Ebenso kann jedes binäre Ereignis firstChild(Label) bzw. nextSibling(Label) durch die Ereignisfolge firstChild(*);self(Label) bzw. nextSibling(*);self(Label) repräsentiert werden.

Dementsprechend benutzen wir – abweichend zu den im Kapitel 2 beschriebenen Umformungsregeln – zur Berechnung der elementaren XPath-Automaten die folgenden Umformungsregeln, welche den Vorteil haben, dass nach einer first-child bzw. next-sibling Achse immer mindestens eine self-Achse steht. Diese Reihenfolge entspricht der Eingabe des Automaten – dem oben beschriebenen, modifizierten binären SAX-Event-Strom.

- $child :: a \rightarrow first-child : *(/self :: */next-sibling :: *)^i/self :: a, 0 \leq i < \infty$
- $following-sibling :: a \rightarrow next-sibling :: *(/self :: */next-sibling :: *)^i/self :: a, 0 \leq i < \infty$
- $descendant :: a \rightarrow first-child : *(/self :: *(/first-child :: *|/next-sibling :: *))^i/self :: a, 0 \leq i < \infty$
- $descendant-or-self :: a \rightarrow self :: a|first-child : *(/self :: *(/first-child :: *|/next-sibling :: *))^i/self :: a, 0 \leq i < \infty$

Zu diesen regulären Ausdrücken bilden wir nun die äquivalenten, nichtdeterministischen Automaten – die sogenannten *elementaren XPath-Automaten* – welche in Abbildung 9.1 gezeigt werden.

9.1.2 Auswertung von Pfad-Anfragen

Der vollständige XPath-Automat zu einer Pfad-Anfrage XP wird konstruiert, indem die elementaren XPath-Automaten zu den Location-Steps von XP in der durch XP vorgegebenen Reihenfolge konkateniert werden. Um die elementaren XPath-Automaten A1 und A2 der Location-Steps L1 und L2 zu einem XPath-Automaten XP zu konkatenieren, wird der Endzustand von A1 mit dem Startzustand A2 zu einem einzigen Zustand zusammengefasst. Der Start-

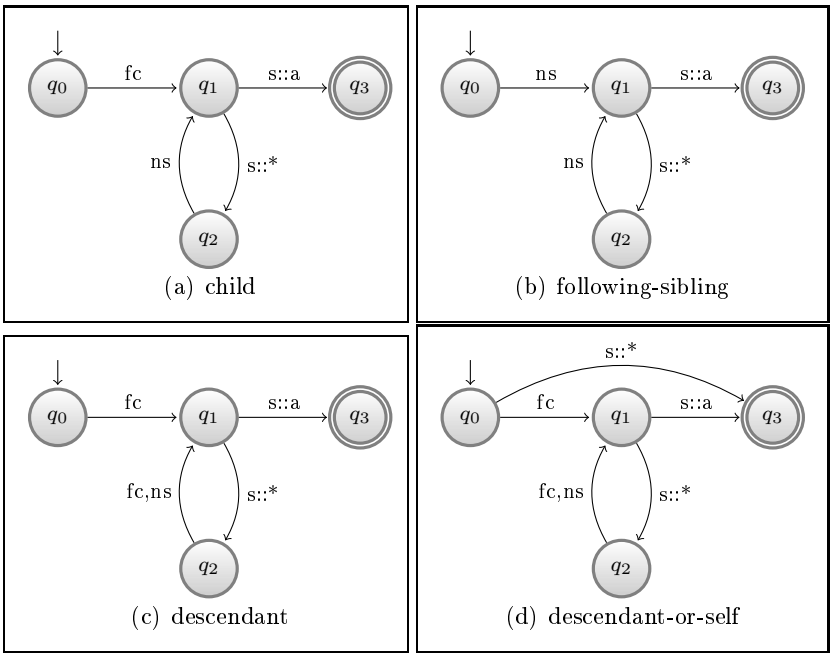

Abbildung 9.1: Elementare XPath-Automaten

zustand von XP ist dann der Startzustand von A1 und der Endzustand von XP ist der Endzustand von A2.

Wird der Endzustand eines Automaten zu einer Pfad-Anfrage erreicht, so wurde im Eingabe-Strom ein Antwort-Fragment gefunden, so dass der dann aktuelle 'Teilbaum' des binären SAX-Stroms als Ergebnis ausgegeben werden kann.

Durch ein binäres SAX-Event firstChild('Name') wird zunächst die Eingabe fc an den Automaten weitergeleitet, und dann wird auf dem Automaten solange die Eingabe s::Name ausgewertet, bis keine Zustandsänderung mehr erfolgt. Hierbei gilt, dass bei Eingabe s::Name sowohl die Übergänge mit Label s::Name, als auch die Übergänge mit Label s::* aktiviert werden.

Beispiel 9.1 *Betrachten wir die XPath-Anfrage XP = /Adressen/Person[./Ort='Berlin']/Name, die nach dem Namen in Berlin lebender Personen fragt. Die Haupt-Pfad-Anfrage ist XP' = /Adressen/Person/Name. Der dazugehörige XPath-Automat wird in Abbildung 9.2 dargestellt.*

Betrachten wir nun als Eingabe dieses Automaten den binären SAX-Strom aus Listing 3.2. Nachdem die ersten 10 Events bis einschließlich des Events firstChild('Person') gelesen und im Automaten ausgewertet wurden, befindet

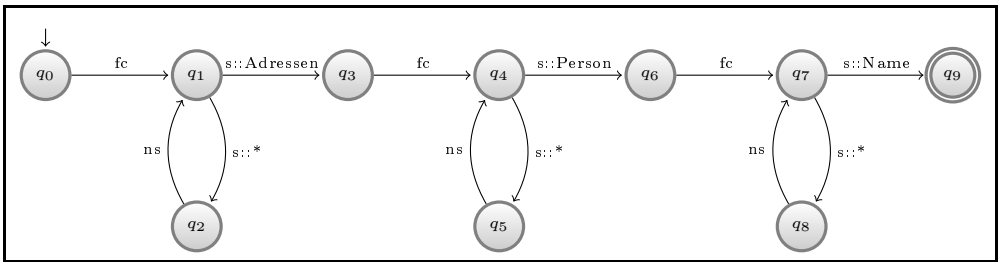

Abbildung 9.2: XPath-Automat zu XP'=/Adressen/Person/Name

sich der Automat in den Zuständen q_5 und q_6. Der Zustand q_6 sagt hierbei aus, dass ein Ergebnis für die Teil-Anfrage /Adressen/Person gefunden wurde, während der Zustand q_5 die Teil-Anfrage /Adressen/ repräsentiert. Zustand q_4 entspricht der Aussage „der Achsentest /Adressen/child:: ist erfüllt".*

Damit q_6 auch wieder nach dem Lesen des später folgenden Events nextSibling('Person') aktiviert wird, muss der Automat sich nach Abarbeiten des Teilbaumes der ersten Person wieder genau in den Zuständen q_5 und q_6 befinden. Hierzu muss das Ende eines Teilbaumes ermittelt werden, was nur mit Hilfe von Zählen der firstChild- und parent-Events geschehen kann. Dies kann jedoch nicht durch einen Automaten geschehen. Daher benutzen wir zusätzlich zum Automaten einen Zustands-Stack, der die Folge der Automaten-Zustände verwaltet. Bevor ein firstChild-Event ausgewertet wird, werden die aktuellen Zustände des Automaten auf dem Zustands-Stack abgelegt. Nachdem ein parent-Event gelesen wurde, wird die oberste Stack-Ebene vom Stack heruntergenommen, und die darin enthaltenen Zustände werden wieder auf dem Stack aktiviert.

Definition 9.2 (XPath-Auswertungs-Stack)**.** Ein XPath-Auswertungs-Stack eines XPath-Automaten XP ist ein 3-Tupel

$$XPE = (XP, \Sigma, \Delta),$$

wobei

- $XP.q_0$ als das initiale Stack-Symbol benutzt wird
- Σ = {fc, ns, p} $\cup$ {s::a | a ist ein Element-Name, '@' gefolgt von einem Attribut-Namen, '=' gefolgt von einer Konstante oder $'*'$} ist die Menge der Eingabe-Symbole
- $\Delta(\Sigma)$ ist eine Auswertungs-Funktion, welche für ein gegebenes Eingabe-Symbol $\sigma \in \Sigma$ eine Folge von Operationen ausführt:
  - $\Delta(fc) = \left\{ \begin{array}{l} \text{push(XP.active);} \\ \text{XP.event(fc);} \end{array} \right\}$

- $\Delta(ns) = \left\{ \text{ XP.event(ns); } \right\}$
- $\Delta(s :: a) = \left\{ \text{ XP.closure(s::a); } \right\}$
- $\Delta(p) = \left\{ \text{ XP.active = pop(); } \right\}$

□

Hierbei feuert die Operation $void\,XP.event(InputSymbol)$ das Ereignis $InputSymbol$ auf dem XPath-Automaten XP. Die Operation $void\,Stack.push(XP)$ legt die aktive Zustandsmenge des XPath-Automaten XP als oberstes Element auf den Stack. Die Operation $void\,Stack.pop()$ löscht das oberste Stack-Element vom Stack und liefert diese als Ergebnis zurück. Der Operator *closure*, welcher bei Eingabe von $s :: a$ ausgeführt wird, sendet wiederholt das Eingabe-Ereignis $s :: a$ an den XPath-Automaten XP, bis sich die Zustandsmenge dieses Automaten nicht mehr ändert.

Bei Eingabe von fc wird also die aktive Zustandsmenge von XP oben auf den Stack gelegt. Anschließend wird für XP das Eingabe-Ereignis fc gefeuert.

Zusammengefasst wird die Auswertung von (filterfreien) Pfad-Anfragen wie folgt durchgeführt: Jede Pfad-Anfrage X wird auf einem binären SAX-Strom BS ausgewertet, indem der XPath-Automat XP zu X berechnet wird, und der XPath-Auswertungs-Stack mit XP als XPath-Automaten und mit BS als Eingabe ausgeführt wird. Jedes binäre SAX-Event wird dem Stack als Eingabe (bzw. als Folge von Eingaben) weitergeleitet, und die Δ-Funktion wird auf dieser Eingabe ausgeführt. Dies führt potentiell sowohl zu Stack-Operationen als auch zu einer Zustandsänderung des XPath-Automaten. Sobald ein Endzustand des XPath-Automaten erreicht wurde, wird der Teilbaum, der durch das aktuelle SAX-Event reprästentiert wird, als Ergebnis z.B. in einen Ausgabestrom geschrieben.

Beispiel 9.2 *Abbildung 9.3 zeigt den Anfang der Auswertung der Anfrage XP'=/Adressen/Person/Name. Der Automat hierzu wird in Abbildung 9.2 dargestellt. Hierbei werden in den Eingabesymbolen aus Platzgründen die Elemente Adressen mit A, Person mit Pe, Name mit N und Postfach mit Po abgekürzt. Als Eingabe wird der binäre SAX-Strom aus Listing 3.2 angenommen, jedoch ohne die 4. Ebene (also die Knoten Name und Postfach enthalten jeweils keinen first-child-Knoten).*

In der Abbildung besteht jeder Knoten aus der aktuellen Liste der aktiven Zustände des Automaten (1. Knotenzeile) sowie dem Zustands-Stack (die weiteren Zeilen).

Würde das Beispiel bis zum Ende durchgeführt, so würde insgesamt 3 Mal der Endzustand q_9 des Automaten erreicht, nachdem das Event s::N eingelesen wurde, welches das jeweilige Namens-Element repräsentiert.

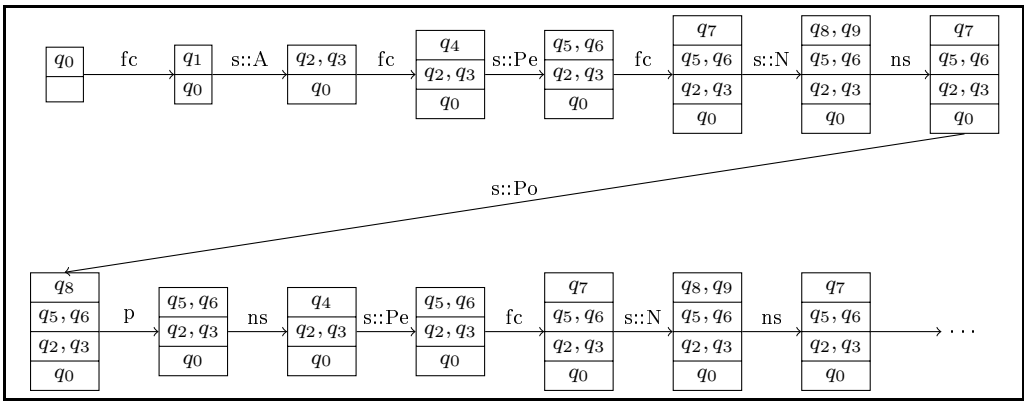

Abbildung 9.3: Ausschnitt der Auswertung der Anfrage /Adressen/Person/Name

9.1.3 Auswertung von Prädikat-Filtern

Enthält ein Location-Step LS einen Prädikat-Filter, so wird auch für den Filter-Pfad ein XPath-Automat gebildet. Dieser wird an den Endzustand des elementaren XPath-Automaten zu LS angehängt. Im Gegensatz zum XPath-Automaten des Haupt-Pfades erzeugt das Erreichen des Endzustandes des XPath-Automaten eines Filter-Pfades keine Ausgabe.

Wird ein Zustand aktiviert, der einen Übergang zum Startzustand eines Filter-Automaten enthält, so wird ein sogenannter *Vorbehalt* erzeugt und an den aktuellen Zustand angehängt. Gleichzeitig wird der Startzustand des Filter-Automaten mit demselben Vorbehalt aktiviert. Dies bedeutet, dass alle eingehenden SAX-Events nicht nur vom Haupt-Automaten, sondern auch von allen aktiven Filter-Automaten ausgewertet werden. Jeder Vorbehalt entspricht einer Boole'schen Variablen, die zu *true* evaluiert wird, sobald ein Endzustand im Filter-Automaten erreicht wird. Entsprechend wird die Boole'sche Variable zu *false* evaluiert, sobald die Filter-Bedingung nicht mehr erfüllt werden kann, also sobald dieser Vorbehalt weder in einem Zustand im Automaten noch in einem auf dem Stack gespeicherten Zustand enthalten ist.

Genauer gesagt werden Vorbehalte also wie folgt berechnet: Seien R, R1 und R2 Mengen von Vorbehalten und sei res: $XP.Q$ x R eine Zuordnung von XPath-Automaten-Zuständen zu Mengen von Vorbehalten. Initial enthält kein Zustand einen Vorbehalt, es gilt also: $\forall q \in XP.Q : res(q, \emptyset)$.

Wird ein Zustand q in $XP.Q$ erreicht, der einen Verweis auf einen Filter-Automaten enthält, so wird die Zuordnung res(q,R) ersetzt durch eine Zuordnung res(q, R $\cup$ {r}), wobei r ein neuer, für q erzeugter Vorbehalt sei.

Wird ein Übergang der Form (q1, Eingabe, q2) gefeuert, werden alle Vorbehalte R1 von q1 mit res(q1,R1) auch dem Zustand q2 hinzugefügt, es gilt also res(q2,R2), wobei R2 = R1 ∪ {r1, ..., rf}, wobei r1, ..., rf neu in q2 erzeugte Vorbehalte seien.

Wird ein Endzustand f des Haupt-Automaten erreicht, der einen Vorbehalt r enthält, der noch nicht zu true oder zu false evaluiert wurde, so wird die Ausgabe der Ergebnis-Fragmente EFr zu f verzögert und in einer Warteschlange verwaltet, bis r ausgewertet wurde. Wird r zu true evaluiert, so wird EFr ausgegeben und aus der Warteschlange entfernt (sobald es den Anfang der Warteschlange erreicht hat). Wird r allerdings zu false evaluiert, so wird EFr aus der Warteschlange gelöscht, ohne ausgegeben zu werden.

Sobald ein Vorbehalt zu false evaluiert wird, verlieren alle Zustände, die diesen Vorbehalt enthalten, ihre Gültigkeit und können im Automaten deaktiviert werden bzw. aus dem Stack gelöscht werden.

Beispiel 9.3 *Erweitern wir Beispiel 9.2 auf die komplette XPath-Anfrage XP = /Adressen/Person[./Ort='Berlin']/Name. Der dazugehörige XPath-Automat wird in Abbildung 9.4 dargestellt. Der Haupt-Automat enthält die Zustände q_0 bis q_9, und der Filter-Automat enthält die Zustände q_{10} bis q_{16} und ist an den Zustand q_6 des Hauptautomaten angehängt.*

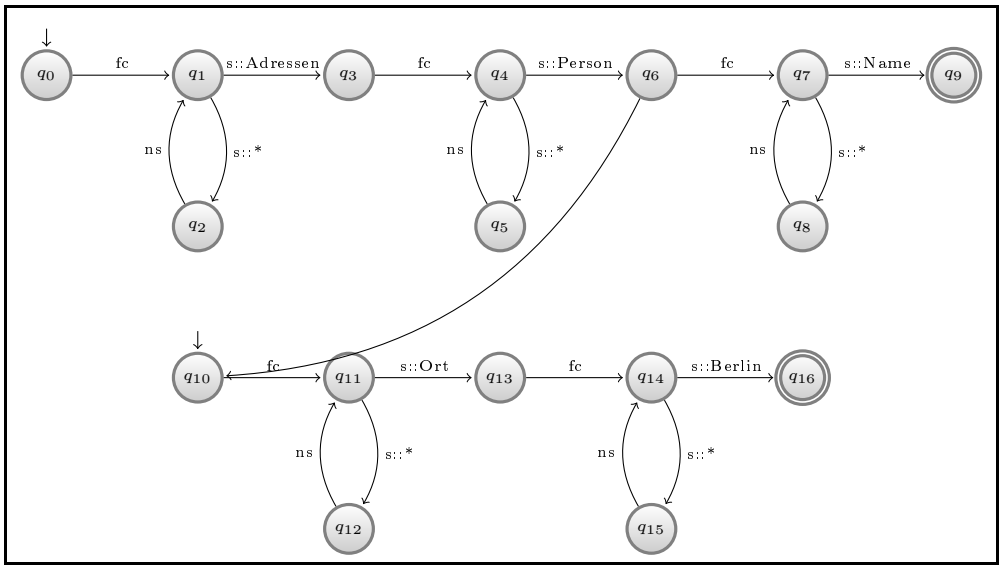

Abbildung 9.4: XPath-Automat zu XP'=/Adressen/Person[Ort='Berlin']/Name

Abbildung 9.5 zeigt die Auswertung des binären Beispiel-Stroms auf diesem Automaten. Bei Aktivierung des Zustandes q_6 wird der Filter-Automat gestartet, und Zustand q_{10} wird aktiv. Gleichzeitig wird durch diesen Aufruf des

Filter-Automaten ein Vorbehalt r_1 angelegt. Im 7. Schritt (Zeile 2, 2. Zustands-Stack) wird der Zustand q_9 erreicht, doch unter dem Vorbehalt r_1. Die Ausgabe im 7. Schritt erfolgt daher noch unter Vorbehalt. Der Vorbehalt r_1 bleibt bestehen, solange r_1 noch nicht evaluiert wurde.

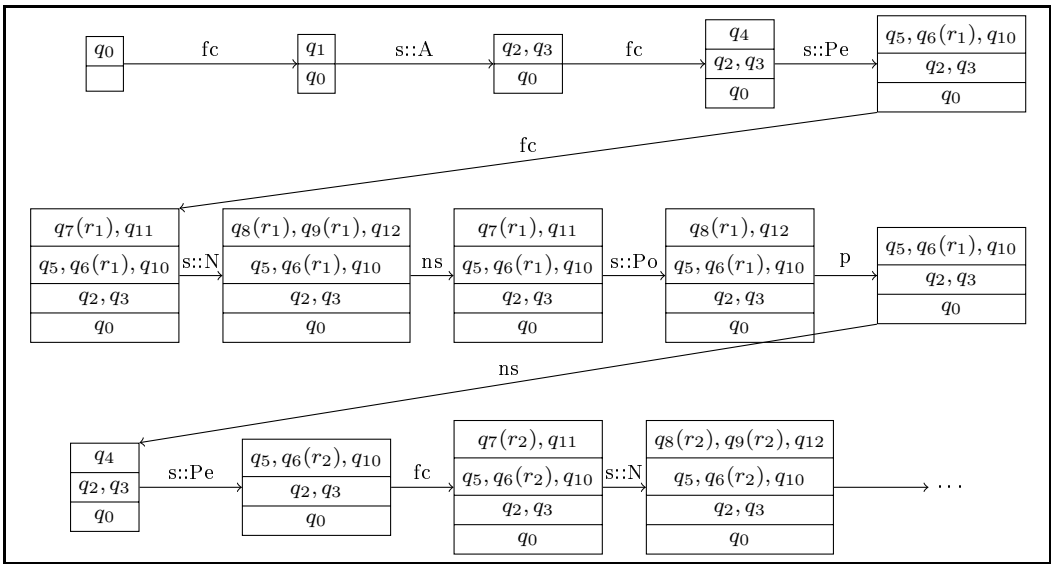

Abbildung 9.5: Ausschnitt der Auswertung der Anfrage /Adressen/Person[Ort='Berlin']/Name

Im 11. Schritt (Zeile 3, 1. Zustands-Stack) schließlich kann r_1 zu false evaluiert werden, da weder ein Zustand existiert, der r_1 enthält, noch ein zu r_1 gehörender Zustand im Filter-Automat mehr auf dem Stack enthalten ist. Somit gehört also das im 7. Schritt ermittelte Ergebnis nicht zur Ausgabe. Im 12. Schritt wird bei erneuter Aktivierung des Zustandes q_6 ein neuer Vorbehalt r_2 erzeugt, und der Filter-Automat wird erneut aktiviert. Dieser Vorbehalt r_2 wird später zu true evaluiert, so dass schließlich als Ergebnis <Name><=Anna Schmidt></Name> ausgegeben werden wird.

9.1.3.1 Zusammengesetzte Prädikat-Filter

Da ein Prädikat-Filter nicht nur einfache Vergleiche Pfad=Wert enthalten kann, sondern eine logische Verknüpfung verschiedener Vergleiche inklusive verschachtelter Negationen, Disjunktionen oder Konjunktionen von Vergleichen, stellen auch Vorbehalte logische Verknüpfungen von Unter-Vorbehalten dar. Ein Prädikat-Filter [(comp1 or comp2) and not comp3], wobei comp1, comp2 und comp3 Vergleiche oder Pfad-Ausdrücke sind, erzeugt einen zusam-

mengesetzten Vorbehalt r = ((r1 or r2) and not r3) und jeweils einen Filter-Automaten für die Unter-Vorbehalte r1, r2 und r3.

Sowohl einfache als auch zusammengesetzte Vorbehalte können mit Hilfe einer Lemma-Tabelle verwaltet werden. Sobald ein Vorbehalt ausgewertet wurde, wird dieses Ergebnis an die Lemma-Tabelle weitergeleitet. Mit Hilfe der Lemma-Tabelle wird dann überprüft, welche Vorbehalte bereits vollständig ausgewertet werden können, und anschließend werden diese Ergebnisse an den XPath-Automaten, den Stack und die Warteschlange weitergeleitet und Zustände sowie Ausgabe-Ereignisse werden gelöscht und evtl. ausgegeben.

9.2 Automaten-basierte XPath-Auswertung mit getFirstChild und getNextSibling

Der bisherige Ansatz – wie im vorangehenden Abschnitt vorgestellt – erhält als Eingabe einen kompletten binären SAX-Strom. Dessen Erzeugung kommt jedoch im Normalfall für alle XML-Repräsentationen einer Dekompression gleich. Daher werde ich nun vorstellen, wie man, basierend auf diesen Konzepten, eine weitere Schnittstelle aufbauen kann, die durch die Verwendung der für die einzelnen Kompressionsverfahren vorgestellten Funktionen first-child und next-sibling das Überspringen nicht benötigter Teilbäume ermöglicht.

Beispiel 9.4 *Betrachten wir den Automaten aus Abbildung 9.4 und nehmen wir an, dass momentan die Zustände q_8 und q_{12} aktiv sind (wie etwa im vorangegangenen Beispiel im 9. Schritt nach Verarbeitung des nextSibling('Postfach')-Events). Wie wir sehen können, werden die beiden Übergänge, die bei q_8 und q_{12} starten, durch das Eingabe-Symbol ns ausgelöst. Dies bedeutet, dass nur das next-sibling 'zielführend' ist, dass also das first-child und der komplette Teilbaum unterhalb des first-childs übersprungen werden können. Dies entspricht einem Aufruf der jeweiligen Funktion next-sibling der XML-Repräsentation.*

Der nun folgende Ansatz ermöglicht genau solch ein Überspringen von nicht relevanten Teilbäumen. Wir gehen hierbei davon aus, dass jede XML-Repräsentation neben den Funktionen firstChild, nextSibling und label auch eine Klasse *XMLNode* implementiert, die aus Informationen besteht, die zusammen einen XML-Knoten im Original-Dokument eindeutig identifizieren. Dies wäre für die Succinct-Darstellung z.B. die Position innerhalb des Bitstroms und für die DTD-Subtraktion das Tupel (KST, n, p) bestehend aus Komprimat KST, Syntaxknoten n und Position p innerhalb des Komprimats. Um das Überspringen nicht relevanter Teilbäume zu ermöglichen, wird in diesem Ansatz der XPath-Auswertungs-Stack des vorherigen Ansatzes durch einen zweiten Stack, den Navigations-Stack, ersetzt. Dieser enthält Paare aus XMLNodes

x und zugehörigen Automaten-Zuständen, die nach Verarbeitung von x aktiv sind.

Im ersten Ansatz war der binäre Strom das steuernde Element: Das Symbol, das als nächstes aus dem SAX-Strom gelesen wurde, entschied über die nächsten auszuführenden Aktionen. Dies ist nun nicht mehr der Fall. Statt dessen sind der Navigations-Stack und der XPath-Automat die steuernden Elemente: Abhängig davon, welche XMLNodes und Zustände oben auf dem Stack liegen bzw. welche Arten von Übergängen von diesen Zuständen ausgehen, werden first-child und next-sibling konsumiert oder übersprungen.

Definition 9.3 (Navigations-Stack)**.** Ein Navigations-Stack eines XPath-Automaten und einer XML-Repräsentation Comp ist ein 3-Tupel

$$XPE = (XP, Comp, navigate()),$$

wobei

- *Comp* eine XML-Repräsentation ist, und *Comp.V* alle Identifizierungs-Informationen zur eindeutigen Identifizierung eines XML-Knotens V beinhaltet. Entsprechend beinhaltet *Comp.root* alle Identifizierungs-Informationen zur eindeutigen Identifizierung des Wurzelknotens des XML-Baums.
- Jede Stack-Ebene aus einem 3-Tupel (node, setOfStates, fc) besteht, wobei node $\in$ Comp.V ein XMLNode von Comp ist, setOfStates $\subseteq$ XP.Q eine Zustandsmenge von XP ist, und fc eine Boole'sche Variable ist.
- Das Tupel (Comp.root, $XP.q_0$, false) das initiale Stack-Symbol ist.
- Die Funktion navigate() die Steuerungs-Funktion ist, welche entsprechend Algorithmus 9.1 definiert ist.

```
1   NavigationStack nst;
2
3   public static void fireEvent(XMLNode x, String event)
        {
4       if(x!=null) {
5           nst.XP.active = states;
6           nst.XP.event(event);
7           Set states = nst.XP.active;
8           if(nst.XP.containsSelf(states)){
9               nst.XP.event('s::' + x.label());
10              states = nst.XP.active;
11          }
```

```
12            nst.push(x,states,false);
13        }
14    }
15
16    public static void navigate()
17    {
18        while(!nst.isEmpty()){
19            StackEntry top = nst.top();
20            XMLNode x = top.getNode();
21            Set states = top.getStates();
22            if(!nst.top().fc){ //first-child noch nicht
                  verarbeitet
23                if(nst.XP.containsFC(states)){
24                    fireEvent(x.getFirstChild(), 'fc');
25                }
26                top.fc=true;
27            }
28            else{
29                 nst.pop();
30                 if(nst.XP.containsNS(states)){
31                    fireEvent(x.getNextSibling(), 'ns');
32                }
33            }
34        }
35    }
```

Algorithmus 9.1: Navigation mit Hilfe des Navigations-Stacks

□

Algorithmus 9.1 beschreibt die Steuerung der XPath-Anfrage-Auswertung durch Stack und Automaten: Wird ein Stack-Eintrag top zum ersten Mal betrachtet, so ist die Boole'sche Variable fc=false (Zeile 22). In diesem Fall wird überprüft, ob es in der zu top gehörenden Zustandsmenge state einen Übergang gibt, der die Eingabe 'fc' verlangt (Zeile 23). Ist dies der Fall, so wird die Methode fireEvent für das first-child aufgerufen (Zeile 24). Zunächst wird überprüft, ob ein first-child existiert (Zeile 4). Nur wenn dies erfüllt ist, wird das Eingabe-Symbol 'fc' an den Automaten gesendet (Zeilen 5-7). Anschließend wird überprüft, ob ein Übergang mit Eingabe 'self' aktivierbar ist (Zeile 8), und, falls dies der Fall ist, wird das self-Event generiert und an den Automaten gesendet (Zeilen 9-10). Anschließend wird das 3-Tupel bestehend aus den first-child-Knoten, der neuen Zustandsmenge und dem Flag fc=false oben

auf den Stack gelegt. Für den Stack-Eintrag top wird das Flag fc=true gesetzt, da nun das first-child abgearbeitet wurde.

Wird ein Stack-Element top zum zweiten Mal betrachtet, so ist das Flag fc=true. Es wird wieder wie oben beschrieben verfahren, nur dass dieses Mal das next-sibling anstelle des first-childs betrachtet wird.

Da für jeden Knoten gilt, dass sein first-child dem next-sibling im Strom vorangeht, garantiert der Stack so ein lineares Durchqueren des Stroms.

Der Automat behandelt Vorbehalte und Ausgabe entsprechend wie im vorangehenden Ansatz. Aus Sicht des Automaten hat sich nichts geändert, da dieser nach wie vor Ereignisse fc, ns und s::x erhält und diese ausführt, lediglich die Anzahl der Ereignisse wurde verringert, da nur noch 'zielführende' Ereignisse ausgeführt werden, alle anderen werden übersprungen.

Beispiel 9.5 *Abbildung 9.6 zeigt einen Beispieldurchlauf des Navigations-Stacks, durchgeführt mit dem Automaten aus Abbildung 9.2 und beschränkt auf die Zustände $q_0 - q_6$, wobei q_6 den Endzustand darstellt. Dies entspricht der XPath-Anfrage XP"=/Adressen/Person.*

Initial befindet sich auf dem Navigations-Stack nur der Wurzelknoten sowie der Startzustand q_0 und das Flag f(alse). Da der Zustand q_0 einen ausgehenden fc-Übergang hat, wird anschließend fc und self::A(dressen) ausgeführt (abgekürzt durch fc::A). Dies führt einerseits dazu, dass das Flag des untersten Stack-Eintrags auf t(rue) gesetzt wird, da das first-child verarbeitet wurde, andererseits wird der Stack-Eintrag (A_1, $\{q_2, q_3\}$, f) auf den Stack gelegt. (Der Index 1 ist hier nur zur besseren Unterscheidung im Beispiel vorhanden.) Die erste Neuerung macht sich im 4. Schritt bemerkbar: Die Zustandsmenge $\{q_5, q_6\}$ enthält keinen fc-Übergang, daher wird weder auf der XML-Repräsentation zum first-child navigiert, noch wird auf dem Automaten ein fc-Event gefeuert. Lediglich das Flag wird auf t(rue) gesetzt, so dass anschließend direkt mit dem next-sibling fortgefahren werden kann. Der komplette, irrelevante Teilbaum unterhalb des Person-Elementes wird somit übersprungen.

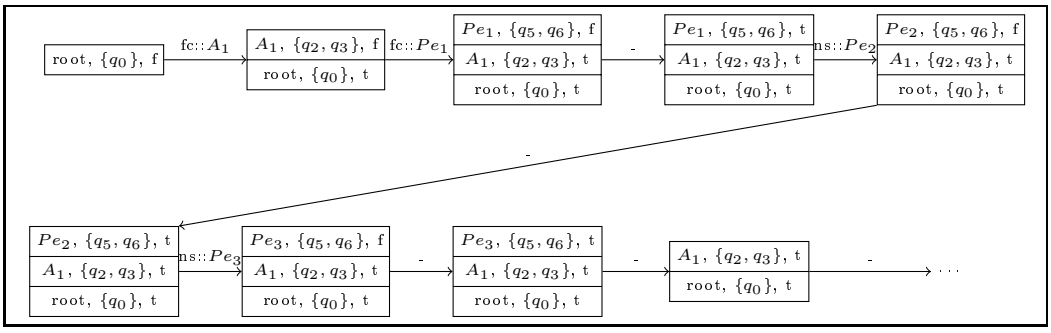

Abbildung 9.6: Ausschnitt der Auswertung der Anfrage /Adressen/Person

9.3 Weitere Optimierungsmöglichkeiten

Der DAG bietet noch eine weitere Optimierungsmöglichkeit: Repräsentiert ein DAG-Knoten mehr als einen XML-Knoten, so können einmal gewonnene Ergebnisse über den zuerst besuchten XML-Knoten auch auf die anderen XML-Knoten übertragen werden, sofern der Automat bei beiden Knoten im selben Zustand ist.

Dies bedeutet, dass man für DAG-Knoten, die mehr als einen XML-Knoten repräsentieren, also die selbst oder mindestens einer deren Vorgängerknoten mehr als eine Eingangskante haben, Tupel bestehend aus Zustand und Ergebnis in einer Lemma-Tabelle speichern kann.

Erreicht man erneut diesen DAG-Knoten, kann man in der Lemma-Tabelle überprüfen, ob für den nun gültigen Automaten-Zustand und diesen DAG-Knoten bereits ein Eintrag vorhanden ist. Ist dieser vorhanden, so kann man die an dieser Stelle gespeicherten Ergebnisse übernehmen und so ein weiteres Durchqueren dieses Teilbaums vermeiden.

9.4 Zusammenfassung: Eigenschaften der Automaten-basierten XPath-Auswertung

In diesem Kapitel wurden zwei Varianten eines Automaten-basierten XPath-Auswertungsansatzes vorgestellt, die alle Vorwärts-Achsen von XPath unterstützen. Wie alle Automaten-basierten Verfahren haben diese beiden Ansätze die Eigenschaft, dass sie die Eingabe – also die XML-Repräsentation (z.B. binärer SAX-Strom oder XML-Komprimat) – linear durchqueren.

Des weiteren ist die Größe der jeweiligen Automaten in O(XP), wobei XP die Anzahl der Location-Steps innerhalb der zu betrachtenden Anfrage ist, da der Automat aus einer Reihe von elementaren XPath-Automaten konkateniert wird, wobei jeder elementare XPath-Automat einem Location-Step der XPath-Anfrage entspricht.

Als weitere Eigenschaft kann der Automat – aufgrund des sehr eingeschränkten Eingabealphabets – sowohl sehr effizient gespeichert als auch ausgeführt werden.

10 Verwandte Arbeiten

In diesem Kapitel werde ich die vorgestellten Ideen mit anderen in der Literatur bereits erwähnten Ideen vergleichen. Analog der in dieser Arbeit vorgestellten Verfahren, konzentrieren sich die im Folgenden erörterten Verfahren im Wesentlichen auf die Struktur-Kompression. Lediglich, wenn eines der Verfahren einen besonderen Ansatz zur Konstanten-Kompression verwendet, wird dies kurz erwähnt. Für einen Überblick über mögliche Ansätze zur Konstanten-Kompression sei ansonsten auf Kapitel 8 verwiesen.

Dieses Kapitel unterteilt sich entsprechend der Hauptbeiträge dieser Arbeit in ein Unterkapitel über XML-Kompression – aufgeteilt in XML-Kodierungen, DAG-Varianten, Schema-basierte Varianten und sonstige Verfahren – sowie in ein Unterkapitel über Anfrage-Auswertung für XML-Datenströme.

10.1 XML-Kompression

10.1.1 XML-Kompression durch platzeffiziente Kodierung

Den ersten Ansatz für ein spezielles Kompressions-Verfahren für XML-Dokumente stellt das in [60] präsentierte Verfahren namens XMill dar. In diesem Verfahren werden Daten und Struktur separat komprimiert. Die Daten werden anhand des umschließenden Element- bzw. Attribut-Namens in Container sortiert, und diese Container werden separat mit einem geeigneten Kompressor komprimiert. Jedem Element und Attribut der Dokument-Struktur sowie jedem Daten-Container wird eine kurze ID zugeordnet. Neben den Daten-Containern existiert eine zweite Datenstruktur, die die Baum-Struktur repräsentiert, und die aus den Element- und Attribut-IDs, die die Start-Tags repräsentieren, den Container-IDs sowie dem Symbol '/', welches einen End-Tag repräsentiert, besteht. Auch diese Datenstruktur wird mit einem generischen Kompressor komprimiert, so dass auf die Baumstruktur nur mit Hilfe von Dekompression zurückgegriffen werden kann, insbesondere kann also nicht direkt auf dem Komprimat navigiert werden.

Der ursprüngliche XMill-Ansatz ist nicht auf unendliche Datenströme anwendbar. Modifiziert man diesen jedoch, so dass immer nur fensterweise komprimiert wird, könnte man mit Hilfe dieser modifizierten Version auch unendliche Datenströme komprimieren, da XMill kein mehrfaches Parsen des XML-Dokuments erfordert.

Die Verfahren XGrind [75], XPRESS [63] und XQueC [8] stellen Erweiterungen des XMill-Ansatzes dar: Die Element- bzw. die Attribut-Namen werden mit Hilfe der Huffman-Kodierung bzw. mit Hilfe einer arithmetischen Kodierung durch kürzere Token dargestellt, und die Daten werden anhand des parent-Elements in Container sortiert. Es wird allerdings auf eine zusätzliche Kompression der Baum-Struktur-Darstellung verzichtet, so dass diese Erweiterungen in der Lage sind, Anfragen direkt auf dem Komprimat auszuwerten. Auch für die Kompression der Daten-Container werden spezielle Kompressoren (wie z.B. ALM [6,7]) eingesetzt, die die Auswertung gewisser Funktionen (z.B. Gleichheits-Tests, Ungleichheits-Tests) direkt auf den komprimierten Konstanten erlauben. Zusätzlich verwendet XQueC eine Art Struktur-DAG als Index auf die komprimierten Daten.

Wie ihr Vorgänger XMill können diese Verfahren auf unendliche Datenströme angewandt werden. Sie erreichen eine deutlich geringere Kompressionsstärke als XMill, erlauben im Gegenzug dazu allerdings die Anfrage-Auswertung direkt auf dem Komprimat.

Auch die Ansätze [13] und [49] basieren im Wesentlichen auf der Darstellung der Element-Namen durch kürzere Tokens. Die Besonderheit von [13] und [49] ist, dass das Komprimat zusätzlich um Informationen angereichert wird, welche eine effizientere Navigation ermöglichen. Dies sind z.B. Anzahl der Kindknoten, Existenz von Text-Inhalten oder Attributen sowie direkte Zeiger zur Position des next-siblings. Beide Verfahren benutzen ein reserviertes Token von 1 Byte Länge, um den End-Tag eines Elements zu kodieren. All diese Verfahren sind auf unendliche Datenströme anwendbar und erlauben die Anfrage-Auswertung direkt auf dem Komprimat.

Der in [79] vorgestellte Ansatz liefert eine weitere Succinct-Darstellung von XML. Auch hier wird nicht die eigentliche Baum-Darstellung von den Element- und Attribut-Namen getrennt, so dass sowohl für die Token, welche Element- und Attribut-Namen repräsentieren, als auch für das Token, welches den End-Tag repräsentiert, 1 Byte, also insgesamt 2 Bytes (im Gegensatz zu insgesamt 1 Token und 2 Bits in dem in Kapitel 4 vorgestellten Ansatz) benötigt werden. Um eine effizientere Navigation auf dem Komprimat zu ermöglichen, reichert der in [79] vorgestellte Ansatz jedes Paket des Komprimats um zusätzliche Informationen (Level des ersten Knotens, minimales und maximales Level innerhalb des Pakets) an. Dies dient dem Überspringen ganzer, irrelevanter Pakete

bei der Navigation zum next-sibling. Diese Erweiterung wäre direkt auf das in dieser Arbeit vorgestellte Verfahren anwendbar.

Einzig das in [48] präsentierte Verfahren, auf dem das in Kapitel 4 dieser Arbeit vorgestellte Verfahren basiert, trennt die Baum-Struktur von den Knoten-Labeln, so dass für die Baum-Struktur eine Liste von öffnenden und schließenden Klammern erzeugt werden kann, die im Wesentlichen dem Bitstrom der in dieser Arbeit erörterten Succinct-Darstellung entspricht. Im Gegensatz zu den in dieser Arbeit vorgestellten invertierten Elementlisten nutzt [48] eine einfache Elementliste, also ein Mapping von öffnenden Klammern zu Token, die jeweils ein Label eines Element- bzw. Attribut-Knotens repräsentieren. Ähnlich wie auch [79] reichert [48] jedes Paket durch einen Index von sogenannten öffenenden und schließenden Pionieren an, die es erlauben, das Paket, in dem sich der End-Tag eines Knotens bzw. der Parent-Knoten eines Knotens befindet, effizient zu ermitteln.

Die in dieser Arbeit vorgestellten invertierten Listen zur Speicherung der Element- und Attribut-Namen bieten einen weiteren Vorteil gegenüber [79]: Neben der effizienteren Speicherung erlauben die invertierten Listen eine effizientere Auswertung der Vorwärtsachsen, da die Anzahl der zu überprüfenden Bitstrom-Positionen mit Hilfe der invertierten Listen eingeschränkt wird, und daher weniger Achsen-Bedingungen überprüft werden müssen.

Im Vergleich zu diesen Verfahren führt die in dieser Arbeit vorgestellte Succinct-Darstellung die bereits in XMill erfolgreich umgesetzte Idee der Trennung zur besseren Kompression fort: Nicht nur Struktur und Daten werden getrennt voneinander komprimiert, sondern auch die Baum-Struktur wird weiter aufgeteilt in die Struktur und die Label. Dies ermöglicht eine noch effizientere Kodierung der Baum-Struktur, wodurch auf eine zusätzliche Komprimierung der Struktur-Daten verzichtet werden kann. Dadurch ist die Succinct-Darstellung in der Lage, Anfragen und Updates direkt auf dem Komprimat zu unterstützen, was z.B. bei XMill nicht möglich ist.

10.1.2 XML-Kompression durch Eliminierung interner struktureller Redundanzen

Der erste Ansatz zur XML-Kompression, welcher auf der Eliminierung interner struktureller Redundanzen (entsprechend DAG-Kompression) basiert, wurde unter dem Namen 'Bisimulation' in [26] vorgestellt. Die Bisimulation stellt den herkömmlichen DAG dar (im Gegensatz zum binären DAG, welcher in dieser Arbeit betrachtet wird), wobei bewiesen wird, dass dieser DAG in linearer Zeit für ein gegebenes XML-Dokument berechnet werden kann. Ebenso wird in [26] die Auswertung der XPath-Vorwärtsachsen sowie deren Komplexität betrachtet. [25] stellt eine Fortführung dieses Ansatzes dar. Einerseits

wird die DAG-Kompression auf der XML-Struktur durchgeführt, während die Text- und Attribut-Werte – dem Grundgedanken von XMill folgend – in verschiedene Container anhand der Pfad-Informationen sortiert werden und dann containerweise komprimiert werden. Für dieses Kompressionsmodell wird des Weiteren die Anfrage-Auswertung der Anfragesprache XQuery erläutert, analysiert sowie evaluiert.

Einen sehr ähnlichen Ansatz verfolgt auch das Verfahren XQZip [34]: Die XML-Struktur wird mit Hilfe der DAG-Kompression zu einem sogenannten *Structure Index Tree (SIT)* komprimiert. Im Gegensatz zu dem in dieser Arbeit vorgestellten Ansatz wird jedoch der herkömmliche DAG erzeugt, wobei nur direkt benachbarte, gleiche Knoten zusammengefasst werden dürfen (und nicht beliebige, gleiche Knoten innerhalb eines vorgegebenen Fensters). Die Text- und Attribut-Werte werden, wie bei XMill, anhand des umgebenden Element- bzw. Attribut-Namens in Container sortiert, die Container in kleinere Blöcke aufgeteilt und Block für Block separat komprimiert. Da bei diesem Ansatz das Augenmerk besonders stark auf effizienter Anfrage-Auswertung für XPath-Anfragen liegt, sind SIT und Container-Blöcke mit Hilfe einer Hash-Tabelle vollständig verzeigert, so dass bei einer Anfrage-Auswertung nur möglichst wenige Blöcke dekomprimiert werden müssen. Im Gegensatz zum in dieser Arbeit vorgestellten Ansatz so wie zu den Ansätzen [25,26] unterstützt XQZip nur die acht XPath-Achsen ancestor, ancestor-or-self, attribute, child, descendant, descendant-or-self, parent und self, nicht aber die „Seitwärtsachsen“ following, following-sibling, preceding und preceding-sibling.

Auch LZCS [5] ist eine Variante des DAG-Verfahrens. LZCS wendet das generische Kompressions-Verfahren LZ77 auf XML-Dokumente an, wobei die kleinste Einheit innerhalb eines XML-Dokuments ein Knoten im XML-Baum ist. Dies bedeutet, dass wiederholt vorkommende Teilbäume durch Rückwärtszeiger ersetzt werden. LZCS entspricht also dem herkömmlichen DAG angewandt auf das gesamte XML-Dokument.

Das in [35] vorgestellte Verfahren stellt eine Weiterentwicklung der Kompression mit Hilfe herkömmlicher DAGs dar. Hier wird nicht nur die stark redundante Struktur von den weniger redundanten Text-Werten getrennt, sondern es wird auch die Struktur analysiert und mit Hilfe einer Heuristik in stark redundante und weniger redundante Anteile zerlegt, so dass das DAG-Verfahren eine höhere Kompressionsstärke erreicht.

BPLEX [28, 44] stellt eine weitere Fortentwicklung der DAG-Kompression dar. Nicht nur gleiche Teilbäume werden mit Hilfe von Rückwärtszeigern zusammengefasst, sondern auch Teilbäume, die einem ähnlichen Muster entsprechen. Hierzu werden die unterschiedlichen Anteile der ähnlichen Teilbäume mit Hilfe von Parametrisierung verallgemeinert und die konkreten Parameter-Werte den Rückwärtszeigern beigefügt. Gibt es zu einem Teilbaum mehr als

eine Möglichkeit der Zusammenfassung mit anderen, ähnlichen Teilbäumen, so wird mit Hilfe einer Heuristik die vielversprechendste Zusammenfassung ausgewählt.

Im Gegensatz zu all diesen Verfahren bietet das in dieser Arbeit vorgestellte DAG-Verfahren eine Überlaufbehandlung für unendliche XML-Datenströme. Während die in diesem Kapitel diskutierten Verfahren bezüglich der XML-Dokumente durch die Größe des verfügbaren Arbeitsspeichers beschränkt sind, kann das in Kapitel 5 dieser Arbeit vorgestellte DAG-Verfahren beliebig große XML-Dokumente und unendliche XML-Datenströme bei konstantem Arbeitsspeicher-Bedarf komprimieren.

Des Weiteren bieten die in dieser Arbeit vorgestellten Kombinationsmöglichkeiten mit dem Succinct-Verfahren bzw. der DTD-Subtraktion eine speichereffiziente Kodierung des DAGs, so dass das DAG-Verfahren nicht nur zur effizienteren Darstellung DOM-ähnlicher Strukturen im Arbeitsspeicher, sondern insbesondere auch zur effizienten XML-Kompression zur Datenübertragung oder zur Speicherung in Dateien genutzt werden kann.

10.1.3 XML-Kompression durch Eliminierung externer Redundanzen

Millau [74] ist das erste Verfahren, welches die DTD zur Verbesserung der Kompression heranzieht. Im Wesentlichen basiert das Verfahren ähnlich wie auch schon XMill auf Ersetzung der Element- und Attribut-Namen durch Token. Die Menge aller Token werden bei Millau jedoch schon vor Beginn der Kompression erzeugt. Hierbei ist ein Token nicht nur eine Repräsentation des Element-Namens, es enthält zusätzlich in den ersten beiden Bits die Information, ob das Element Attribute und Inhalt enthält. In dieser Variante werden nicht alle aufgrund der DTD redundanten Informationen aus dem XML-Dokument entfernt; z.B. die Zusammenhänge der Sibling-Knoten untereinander, die durch die DTD gegeben werden, werden bei der Kompression durch Millau nicht berücksichtigt.

In einer zweiten Variante von Millau, genannt DDT Compression (Differential DTD Tree Compression), [74] werden ähnlich wie bei der DTD-Subtraktion nur Informationen für die DTD-Operatoren '?', '|', '+' und '*' gespeichert. Methodisch werden dazu die DTD als Graph und das XML-Dokument als DOM-Baum aufgebaut und simultan durchquert und verglichen. Durch die Betrachtung des XML-Dokuments als DOM-Baum ergeben sich allerdings erhebliche Nachteile durch den durch DOM verursachten Hauptspeicher-Verbrauch, wodurch laut [74] ein Dokument mit 288735 Bytes bereits zu Problemen bei der Kompression führte.

Auch in XCQ [66] werden zu den DTD-Operatoren '?', '|', '+' und '*' zusätzliche Informationen gespeichert, die es erlauben, die Struktur des XML-Dokumentes zu speichern: 1 Bit für den unären '?'-Operator, 1 Bit für den binären '|'-Operator, n+1 (bzw. n) Bits für den unären '*'-Operator (bzw. für den unären '+'-Operator), wobei ein 1-Bit für eine weitere Wiederholung und ein 0-Bit für das Ende der Wiederholungskette steht. Desweiteren benutzt XCQ die DTD, um vor Durchführung der Kompression die Menge aller Pfade zu berechnen und für jeden Pfad einen separaten Daten-Container zur späteren Daten-Kompression entsprechend des XMill-Konzeptes bereitzustellen. Aufgrund dessen ist XCQ auf nicht-rekursive DTDs beschränkt, da andernfalls die Menge aller durch die DTD erlaubten Pfade unbegrenzt groß wäre.

Die Verfahren XAUST [73], XENIA [76] und [59] arbeiten Automaten-basiert: Aus dem gegebenen Schema (DTD bei [59], XML Schema bei XENIA [76], bzw. RelaxNG bei XAUST [73]) wird ein Automat generiert. Sobald ein Zustand mehr als eine ausgehende Transition hat, werden diese Transitionen mit minimaler Bit-Anzahl durchnummeriert. Der Automat konsumiert das XML-Dokument als Eingabe. Sobald eine Transition gefeuert wird, an die eine Bitfolge angehängt wurde, wird die entsprechende Bitfolge ins Komprimat geschrieben. Dadurch wird ein sehr ähnliches Komprimat wie bei XCQ erreicht, da auch binäre Entscheidungen für '|' und '?' mit je 1 Bit kodiert werden und für die '*'- und '+'-Operatoren n+1 bzw. n Bits kodiert werden müssen (je 1 Bit pro Wiederholung zzgl. 1 Bit für das Ende der Wiederholungen).

Mit Ausnahme von Millau, das nur sehr kleine Dokumente komprimieren kann, aufgrund der methodischen Schwäche, dass der gesamte DOM-Baum des XML-Dokumentes in den Arbeitsspeicher geladen werden muss, sind all diese Verfahren ebenso wie DTD-Subtraktion auf unendliche Datenströme anwendbar. Alle vorgestellten Verfahren erlauben, ebenso wie DTD-Subtraktion, Anfrage-Auswertung sowie Updates direkt auf den komprimierten Daten.

Betrachtet man jedoch die Kodierung der Entscheidungen, so benötigt DTD-Subtraktion für die binären Entscheidungen für die Operatoren '|' und '?' ebenso wie die anderen Verfahren 1 Bit. Für die Operatoren '*' und '+' werden jedoch von allen anderen Verfahren für n Wiederholungen ca. n Bits benötigt, während DTD-Subtraktion diese mit Hilfe einer statischen Huffman-Kodierung mit deutlich weniger als n Bits kodiert (ca. 0,7*n Bits im Durchschnitt für Wiederholungsanzahlen n von 1-24). So ist insbesondere bei großen Dateien, also bei Dateien mit vielen gleichnamigen Siblings, welche in der DTD durch einen '*'- oder '+'-Operator repräsentiert werden, die zu erwartende Kompressionstärke der XML-Struktur von DTD-Subtraktion höher als die Kompressionsstärke der anderen in diesem Kapitel vorgestellten Verfahren.

10.1.4 Weitere XML-Kompressions-Verfahren

Das in [43] vorgestellte Verfahren entspricht vom Grundkonzept her keinem der in dieser Arbeit vorgestellten Verfahren. Das Konzept basiert auf der Burrows-Wheeler-Transformation. Die XML-Daten werden also so transformiert, dass andere Kompressions-Verfahren auf den transformierten Daten eine stärkere Kompression erreichen als auf den ursprünglichen Daten. Aufgrund der Transformation ist dieser Ansatz nicht auf unendliche Datenströme anwendbar. Das Verfahren erlaubt die Anfrage-Auswertung direkt auf dem Komprimat, sofern das Komprimat um einige Index-Informationen angereichert wird.

Das in [33] vorgestellte Verfahren XMLPPM basiert auf einem probabilistischen Konzept. Es basiert auf der Annahme, dass innerhalb eines Kontextes das bislang am häufigsten aufgetretene Element auch das in der Zukunft am wahrscheinlichsten auftretende Element ist. Daher wird das im aktuellen Kontext bislang häufigste Element mit einem kurzen Token dargestellt. Dementsprechend repräsentiert ein Token nicht durchgängig das selbe Element, sondern, welches Element durch ein Token repräsentiert ist, hängt einerseits vom aktuellen Kontext, andererseits auch von den bisher gelesenen Elementen innerhalb dieses Kontextes ab. Weil ein Element zu einem Token nur ermittelt werden kann, wenn das komplette Dokument vor dieser Stelle gelesen wurde, unterstützt dieses Verfahren keine Anfrage-Auswertung auf dem Komprimat. Es kann auf unendliche Datenströme angewandt werden, da ein einmaliges, lineares Durchqueren der Daten genügt.

10.2 Effiziente XPath-Auswertung auf XML-Datenströmen

Es existieren bereits verschiedene Ansätze zur Auswertung von XPath-Anfragen auf XML-Datenströmen. Sie können hauptsächlich aufgrund des unterstützten XPath-Sprachumfangs kategorisiert werden. Nahezu alle dieser Verfahren basieren auf Automaten (X-scan [56], XMLTK [9], YFilter [41], [51], [52], AFilter [30], XSQ [71], SPEX [24, 67]) oder Syntaxbäumen ([10], $\chi\alpha o\varsigma$ [12], [31], [32]).

Der Unterschied von Automaten und Syntaxbäumen liegt in der Steuerung: Während beim Automaten die Eingabe, also der XML-Strom, die steuernde Instanz ist, ist beim Syntaxbaum der Baum, also die Anfrage, die steuernde Instanz.

All diese Ansätze unterstützen die Achsen child und descendant-or-self, und viele dieser Ansätze unterstützen Prädikat-Filter und Wildcards, aber im Gegensatz zu dem in Kapitel 9 dieser Arbeit präsentierten Ansatz unterstützt keiner dieser Ansätze die sibling-Achsen.

X-scan [56], XMLTK [9], and YFilter [41] unterstützen die child- und die descendant-or-self-Achse sowie Wildcards, indem sie einen endlichen Zustandsautomaten nutzen. [51] (für den Haupt-Pfad) und [52] (für die Prädikat-Filter) verwenden einen deterministischen, endlichen Automaten (DFA) in einer sogenannten 'lazy'-Fassung, das heisst, dass der DFA nicht vor Beginn vollständig erzeugt wird, sondern es werden weitere Zustände nur bei Bedarf hinzugefügt. AFilter [30] ist ein anpassbarer Ansatz zur XPath-Anfrage-Auswertung, welcher eine minimale Grundanforderung bzgl. des Speichers stellt, und der linear in Anfrage- und Datengröße skaliert. Sollte mehr Speicher zur Verfügung gestellt werden, so nutzt AFilter den verbleibenden Speicher für einen Caching-Ansatz, um die Anfragen schneller beantworten zu können. Ähnlich wie YFilter [41], wurde AFilter entworfen, um große Mengen von Queries auszuwerten.

XSQ [71] und SPEX [24, 67] nutzen eine Hierarchie oder ein Netzwerk von Transducern, das heisst, sie nutzen Automaten, die um einen Puffer erweitert wurden und deren Zustände um Aktionen erweitert wurden, um XPath-Anfragen auszuwerten. Der von XSQ unterstützte XPath-Sprachumfang umfasst Prädikat-Filter, wobei höchstens ein Prädikat-Filter pro Location-Step erlaubt ist, und die Prädikat-Filter lediglich Pfad-Wert-Vergleiche mit Pfaden der Länge 1 bestehend aus den Achsen child, text oder attribute enthalten dürfen. Das Konzept basiert auf je einem nicht-deterministischen Push-Down Transducer (PDT) pro Location-Step. Diese Transducer werden dann zu einer Hierarchie zusammengefasst.

[57] diskutiert die Auswertung der child- und descendant-or-self-Achsen inklusive Prädikat-Filtern (inklusive Funktionen und Arithmetik) und Wildcards in XQuery unter Verwendung von TurboXPath. Die Eingabe-Anfrage wird in eine Menge von Syntaxbäumen transformiert. Wird eine Entsprechung eines Syntaxbaumes innerhalb des Datenstroms gefunden, so werden die zugehörigen Werte in Form eines Tupels gespeichert, um später bezüglich der Prädikat- und Join-Bedingungen getestet zu werden. Die Ausgabe besteht dann später aus denjenigen Tupeln, die die entsprechenden Bedingungen erfüllt haben.

[10] und $\chi\alpha o\varsigma$ [12] erzeugen zunächst ebenfalls einen Syntaxbaum (zuzüglich eines Syntax-DAGs in [12], da diese somit zusätzlich die parent- und ancestor-Achse unterstützen). Der Syntaxbaum wird genutzt, um die nächsten relevanten Knoten, sowie deren Ebene innerhalb des XML-Baums vorherzusagen. Betrachten wir z.B. die Anfrage //a/b und einen Treffer für 'a' in Ebene 3. Dann wäre der nächste relevante Knoten ein Knoten mit Label 'b' in Ebene 4.

[31] stellt einen weiteren, hauptsächlich auf Syntaxbäumen basierenden, Ansatz dar. In diesem Ansatz werden jedoch die Syntaxbäume zu einem Prefix-Trie wie folgt zusammengefasst: Gleiche Prefix-Folgen von child-Achsen Loca-

tion-Steps verschiedener Queries werden zu einem einzigen Pfad innerhalb des Prefix-Tries zusammengefasst.

Der in [32] vorgestellte Ansatz verwendet eine Struktur, die einem Syntaxbaum mit je einem Stack pro Knoten ähnelt. Diese Stacks werden verwendet, um XML-Knoten zu speichern, die ein Ergebnis einer durch den Syntaxknoten repräsentierten Teil-Anfrage darstellen (oder die – im Falle von einschränkenden Prädikat-Filtern – mögliche Ergebnisknoten darstellen).

Im Gegensatz zu all diesen Ansätzen unterstützt der in dieser Arbeit vorgestellte Ansatz zusätzlich die Achsen following und following-sibling. Des Weiteren unterstützt er – im Gegensatz zu [57] und [71] – rekursive XML-Daten, also Daten, in denen derselbe Element-Name innerhalb eines child-Pfades wiederholt vorkommen kann.

11 Evaluierung der vorgestellten Ansätze

In diesem Kapitel werde ich die vorgestellten Verfahren zur Kompression hinsichtlich Kompressionsstärke sowie Kompressions-, Dekompressions- und Anfrage-Auswertungszeit untereinander verglichen.

11.1 Messumgebung

Alle Messungen wurden auf einem Intel Pentium M mit 1300 MHz und 768 MB Arbeitsspeicher unter dem Betriebssystem Windows 2000 durchgeführt. Für die im Rahmen dieser Arbeit entwickelten Verfahren wurde Java 1.6 als Implementierungs-Sprache eingesetzt.

Zur Evaluierung der Verfahren wurden die folgenden Test-Dokumente verwendet:

- XMark – durch den XMark-Benchmark [72] erzeugtes XML-Dokument, welches Auktionsdaten enthält. Je nach Skalierungsfaktor variiert die Größe. So bedeutet ein Skalierungsfaktor von 0,001 116 kB, ein Skalierungsfaktor von 0,01 1,2 MB, ein Skalierungsfaktor von 0,1 11,3 MB und ein Skalierungsfaktor von 1 113 MB. Ist kein weiterer Faktor angegeben, so beträgt die Dokumentgröße 5,3 MB.
- Hamlet (0,3 MB) – eine XML Version des bekannten Shakespeare-Schauspiels.
- Catalog-01 (10,6 MB), Catalog-02 (105,3 MB), Dictionary-01 (10,8 MB), Dictionary-02 (106,4 MB) – durch den XBench-Benchmark [78] erzeugte XML-Dokumente.
- DBLP(308,2 MB) – eine Sammlung bibliographischer Informationen zu wissenschaftlichen Publikationen im Bereich Informatik.

11.2 Kompression

Hinsichtlich Kompressionsrate sowie Kompressions- und Dekompressionszeit wurden die in der Arbeit vorgestellten Verfahren mit den folgenden frei verfügbaren Kompressoren verglichen:

- XMill [60] – ein XML-Kompressor, welcher die Struktur-Knoten mit Hilfe von Token darstellt und die Konstanten entsprechend der umgebenden Element- und Attributnamen in komprimierte Container sortiert. Zur Kompression der Container stehen unter anderem GZip und BZip2 zur Verfügung.
- GZip – ein generischer Textkompressor basierend auf LZ77 und Huffman-Kodierung.
- BZip2 – ein generischer Textkompressor basierend auf der Burrows-Wheeler-Transformation.

Im Gegensatz zu den in dieser Arbeit vorgestellten Verfahren erlauben diese drei Kompressoren keine Anfrage-Auswertung direkt auf dem Komprimat. Soll eine Anfrage ausgewertet werden, so muss das Komprimat zunächst dekomprimiert werden, und das Ergebnis der Anfrage muss anschließend wieder komprimiert werden.

Hierbei ist zu beachten, dass für XMill eine native Anwendung zur Verfügung stand, während alle anderen Verfahren in Java implementiert sind und unter Java 1.6 ausgeführt wurden. Dies beeinträchtigt möglicherweise die Vergleichbarkeit der Zeitmessungen.

Da der DAG sich ohne eine geeignete Kodierung – wie z.B. durch Kombination mit der Succinct-Darstellung oder der DTD-Subtraktion – nicht als Datei-Kompressor eignet, sondern lediglich als DOM-Variante mit verringerter Knoten- und Kanten-Anzahl, wurde das reine DAG-Verfahren nicht mit den anderen XML-Kompressoren verglichen, sondern statt dessen wurden die hybriden Varianten Succinct+DAG und DTD-Subtraktion+DAG zur Evaluierung herangezogen. Um dennoch den Einfluss des DAGs messen zu können, wurde in einer ersten Messung die Struktur des DAGs mit der ursprünglichen XML-Struktur verglichen.

Die Messungen hinsichtlich Kompressionsrate, Kompressions- und Dekompressionszeit gliedern sich wie folgt:

- Vergleich der Strukturkompression der vier Verfahren Succinct, Succinct + DAG, DTD-Subtraktion und DTD-Subtraktion + DAG.
- Untersuchung der Skalierung der Strukturkompression der vier Verfahren Succinct, Succinct + DAG, DTD-Subtraktion und DTD-Subtraktion + DAG.

- Vergleich der dazu unabhängig kombinierbaren Konstanten-Kompressions-Varianten GZip und BZip2 ohne getrennt komprimierte Container sowie GZip und BZip2 mit getrennt komprimierten Daten-Containern je umgebendem Element- und Attributnamen (entsprechend XMill).
- Vergleich der Gesamtkompression der vier Verfahren Succinct, Succinct + DAG, DTD-Subtraktion und DTD-Subtraktion + DAG in Kombination mit GZip inklusive getrennt komprimierter Daten-Container mit GZip und GZip-basiertem XMill.
- Vergleich der Gesamtkompression der vier Verfahren Succinct, Succinct + DAG, DTD-Subtraktion und DTD-Subtraktion + DAG in Kombination mit BZip2 inklusive getrennt komprimierter Daten-Container mit BZip2 und BZip2-basiertem XMill.

11.2.1 Kompressionsrate der Struktur-Kompression

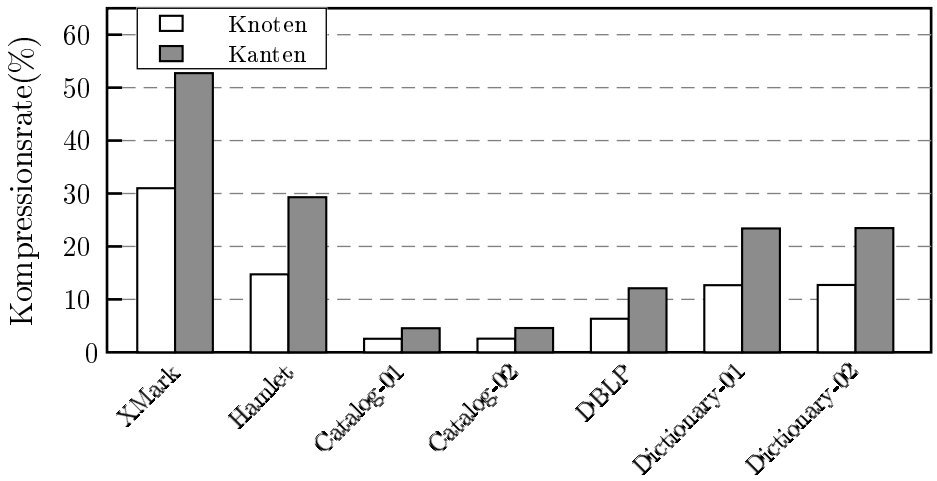

Abbildung 11.1: Vergleich DAG zu binärem XML-Baum

Abbildung 11.1 zeigt das Größenverhältnis des binären DAGs zum ursprünglichen XML-Baum. Je nach Dokument variiert die Kompressionsstärke des DAGs: Die stärkste Kompression erreicht der DAG beim Dokument Catalog-01 mit 2,5% der Knoten des XML-Baums und 4,6% der Kanten, die schwächste Kompression erreicht der DAG bei XMark mit 31% der Knoten und 52,7% der Kanten.

Es ist weiterhin zu beachten, dass das Verhältnis Knoten-Kompression zu Kanten-Kompression schwankt, was mit der durchschnittlichen Größe der wiederverwendeten Teilbäume zusammenhängt: Wird z.B. ein Teilbaum, beste-

hend aus einem Knoten, wiederverwendet, so enthält der DAG einen Knoten weniger, jedoch keine Kante weniger als der XML-Baum.

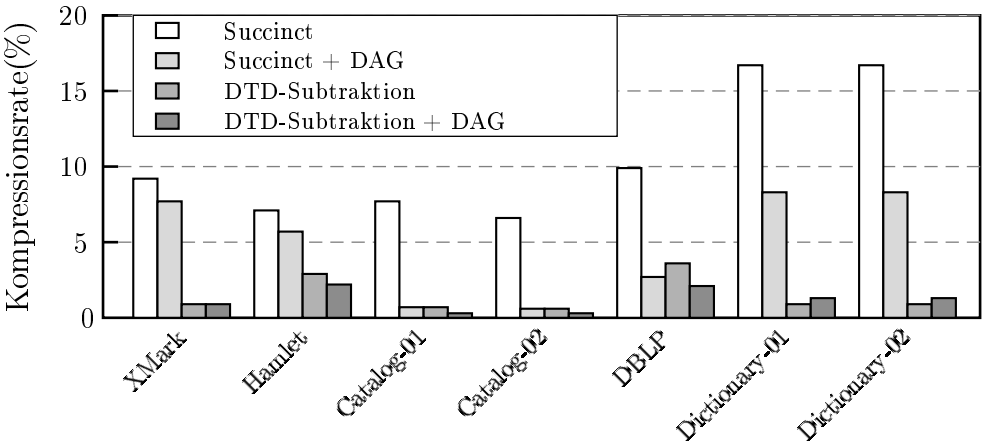

Abbildung 11.2: Kompressionsvergleich der Struktur-Kompression

Abbildung 11.2 zeigt die Struktur-Kompression der vier Verfahren Succinct, Succinct + DAG, DTD-Subtraktion und DTD-Subtraktion + DAG. Das Kompressions-stärkste Verfahren ist hierbei DTD-Subtraktion + DAG, welches Kompressionsraten von 0,3% bis 2,2% auf der Dokument-Struktur erreicht, das Kompressions-schwächste Verfahren ist die reine Succinct-Darstellung, welche Kompressionsraten von 6,6% bis 16,7% erreicht. Mit Ausnahme des Dokumentes DBLP gilt die Aussage, dass die Succinct-basierten Verfahren schwächer komprimieren als die DTD-Subtraktion-basierten Verfahren. Ebenso gilt – mit Ausnahme der Dateien Dictionary-01 und -02 – dass die DAG-Varianten stärker komprimieren als die reinen Verfahren.

Es ist zu beobachten, dass die Kombination Succinct + DAG eine stärkere Verbesserung gegenüber der reinen Succinct-Variante erreicht als die Kombination DTD-Subtraktion + DAG gegenüber der reinen DTD-Subtraktion. Dies liegt daran, dass der relative Speicherbedarf eines DAG-Zeigers in der DTD-Subtraktion deutlich höher ist als in der Succinct-Darstellung. Dies führt dazu, dass in der DTD-Subtraktion weniger Verweise realisiert werden, da Verweise auf kleine Teilbäume teurer sind als das Wiederholen des Teilbaums. Daher kann die Kombination DTD-Subtraktion + DAG im Vergleich zur DTD-Subtraktion nur eine deutlich schwächere Verringerung der Kompressionsrate erreichen als die Kombination Succinct + DAG im Vergleich zu reinem Succinct.

Die Skalierung der Struktur-Kompression der vier Verfahren wird in Abbildung 11.3 anhand des XMark-Benchmarks dargestellt, wobei die erzeugten Dateien mit den Skalierungs-Faktoren 0,001, 0,01, 0,1 und 1 erzeugt wurden.

Für alle Verfahren gilt, dass die Kompressionsrate zunächst mit steigender Dokumentengröße leicht absinkt, um dann nahezu konstant zu bleiben. Das Absinken zu Anfang ist damit zu erklären, dass gewisse Daten nur einmalig zu Anfang des Komprimats geschrieben werden müssen. Je größer die Datei ist, desto kleiner ist der Anteil dieser Daten am Struktur-Komprimat.

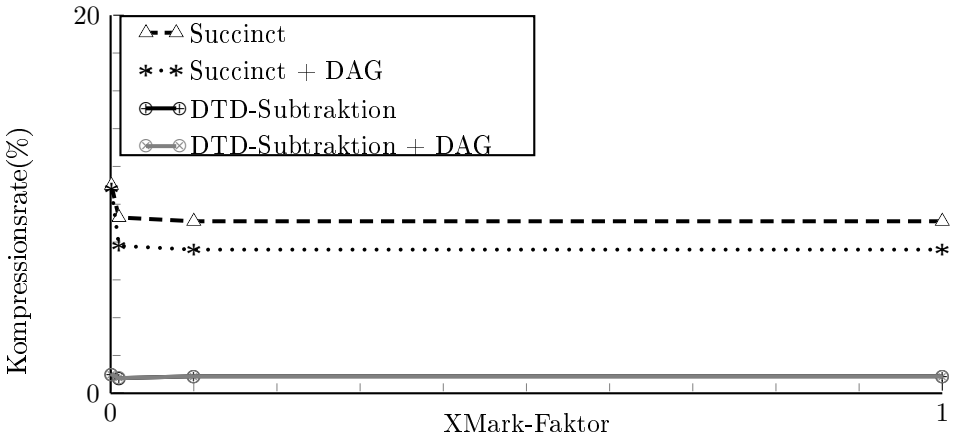

Abbildung 11.3: Skalierung der Kompressionssrate

11.2.2 Kompressionsrate der Konstanten-Kompression

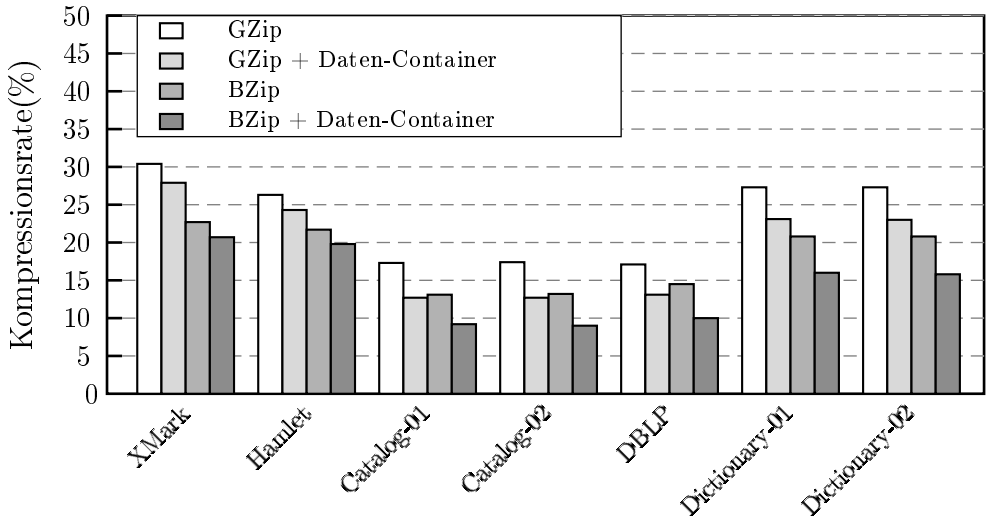

Abbildung 11.4: Kompressionsvergleich der Konstanten-Kompressionen

Abbildung 11.4 zeigt den Vergleich von vier Varianten zur Konstanten-Kompression: GZip und BZip2 ohne getrennt komprimierte Container sowie GZip

und BZip2 mit getrennt komprimierten Daten-Containern je umgebendem Element- und Attributnamen. Hierbei ist zu erkennen, dass bezüglich der Kompressionsstärke die BZip2-basierten Varianten den jeweiligen GZip-basierten Varianten überlegen sind, und die Varianten mit getrennten Daten-Containern den Varianten ohne Trennung überlegen sind.

Für die weiteren Vergleiche mit den Kompressoren XMill, GZip und BZip2 wurden daher die Struktur-Kompressions-Verfahren mit den Varianten kombiniert, die getrennt komprimierte Container verwenden. Erstens erreichen diese eine stärkere Kompression, zweitens ist so auch eine bessere Vergleichbarkeit mit XMill gegeben, welches auch diese Idee zur Konstanten-Kompression verfolgt.

11.2.3 Gesamt-Kompressionsrate

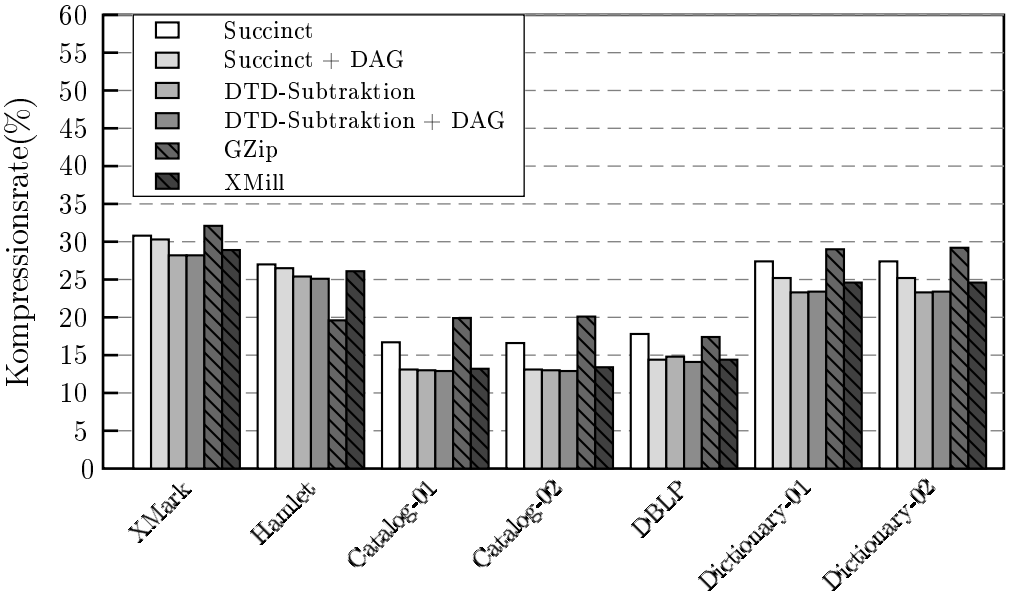

Abbildung 11.5: Kompressionsvergleich der Gesamtkompression GZip-basierter Kompressoren

Abbildungen 11.5 und 11.6 zeigen einen Vergleich der Gesamtkompressionsstärke mit anderen GZip- bzw. BZip2-basierten Kompressoren. In Abbildung 11.5 wurden die vier in dieser Arbeit vorgestellten Struktur-Kompressions-Verfahren mit GZip + Daten-Container als Konstanten-Kompressor kombiniert, in Abbildung 11.6 mit BZip2 + Daten-Container. Jeweils das gleiche Konstanten-Kompressions-Verfahren wurde auch für XMill ausgewählt.

Bei den GZip-basierten Verfahren zeigt sich, dass – ausser beim kleinsten Dokument Hamlet – alle vier Verfahren stärker komprimieren als GZip selbst. Des weiteren zeigt sich, dass die auf DTD-Subtraktion basierenden Verfahren stärker komprimieren als XMill, die auf Succinct basierenden Verfahren jedoch ein wenig schwächer.

Auch bei den BZip2-basierten Verfahren komprimieren die auf DTD-Subtraktion basierenden Verfahren im Allgemeinen stärker als XMill, während XMill stärker komprimiert als die auf Succinct basierenden Verfahren. Im Gegensatz zu GZip komprimiert BZip2 jedoch im Allgemeinen stärker als die reine Succinct-Variante, es komprimiert jedoch schwächer als die Verfahren Succinct + DAG (mit Ausnahme von XMark und Hamlet), DTD-Subtraktion sowie DTD-Subtraktion + DAG.

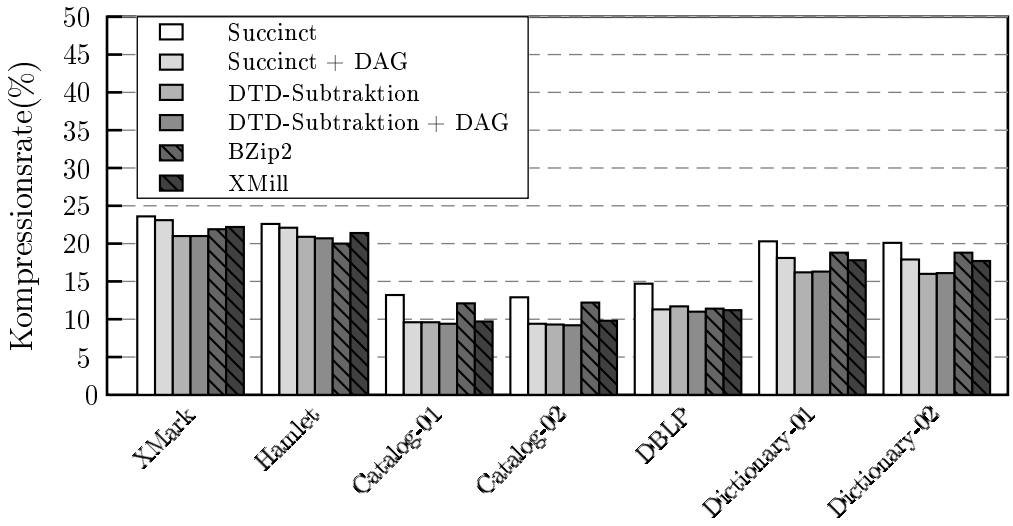

Abbildung 11.6: Kompressionsvergleich der Gesamtkompression BZip2-basierter Kompressoren

11.2.4 Kompressionszeit der Struktur-Kompression

Hinsichtlich der Kompressionszeit zeigt sich ganz klar der Trade-Off im Vergleich zur Kompressionsrate: diejenigen Verfahren, welche eine höhere Kompressionsrate erreichen, benötigen dementsprechend auch mehr Zeit, die Kompression zu berechnen.

Abbildung 11.7 zeigt den Vergleich der Kompressions-Durchsätze der vier vorgestellten Struktur-Kompressions-Verfahren untereinander. Hierbei erreichen die Succinct-basierten Verfahren höhere Durchsätze – sind also schneller – als die DTD-Subtraktion-basierten Verfahren. Ebenso erreichen die XML-basierten Verfahren höhere Durchsätze als die DAG-basierten. Das schnellste

Verfahren hierbei ist das Succinct-Verfahren mit Durchsätzen zwischen 2947 Bytes/ms (bzw. 22,5 Mbit/s) und 5699 Bytes/ms (bzw. 43,5 Mbit/s), das langsamste Verfahren ist DTD-Subtraktion + DAG mit Durchsätzen zwischen 582 Bytes/ms (bzw. 4,4 Mbit/s) und 1895 Bytes/ms (bzw. 14,5 Mbit/s). Im Vergleich dazu hat ADSL eine maximale Empfangsrate von 8 Mbit/s.

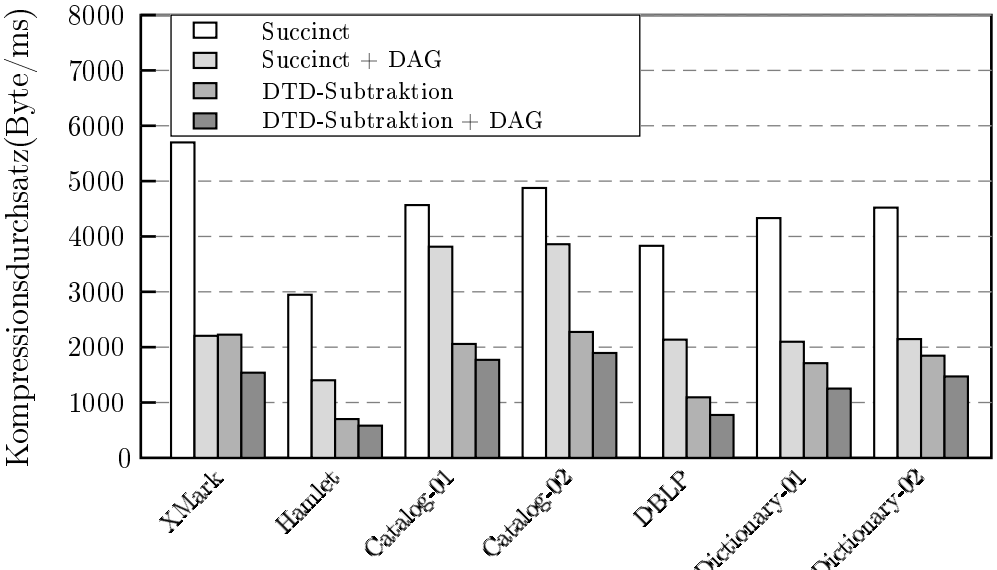

Abbildung 11.7: Vergleich der Kompressions-Durchsätze der Struktur-Kompression

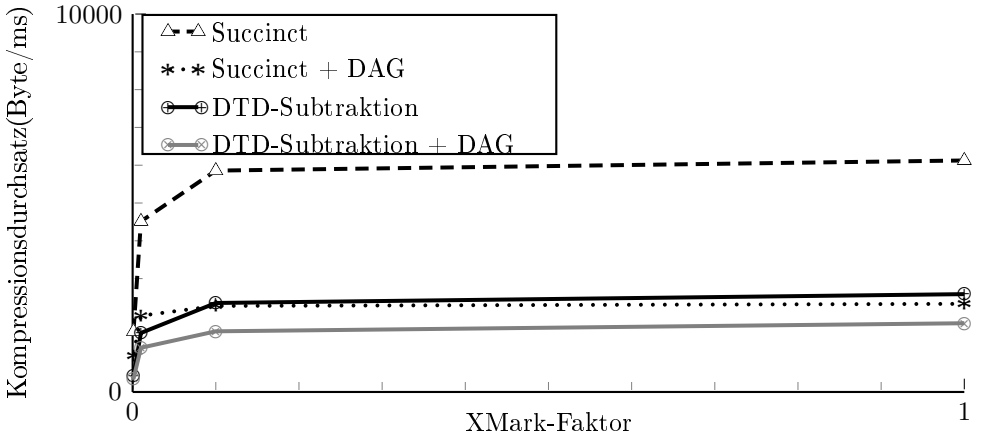

Abbildung 11.8: Skalierung des Kompressionsdurchsatzes

Abbildung 11.8 zeigt die Skalierung der vier Verfahren hinsichtlich der Kompressionszeit. Auch hier zeigt sich, dass mit steigender Dokumentgröße der

Durchsatz zunächst leicht steigt, dann jedoch nahezu konstant bleibt. Dies ist wieder dadurch zu erklären, dass gewisse Initialisierungsschritte nur einmalig pro Kompression unternommen werden müssen und der Anteil dieser Schritte an der Gesamtzeit prozentual mit steigender Dokumentgröße sinkt.

11.2.5 Kompressionszeit der Konstanten-Kompression

Auch bei der Konstanten-Kompression (Abbildungen 11.4 und 11.9) zeigt sich der Trade-Off zwischen Kompressionsstärke und Kompressionszeit. Hier zeigt sich GZip als schnellster Kompressor, gefolgt von GZip + Daten-Container, gefolgt von BZip2, während BZip2 + Daten-Container der langsamste Kompressor ist. Die Durchsätze schwanken hierbei von 164 Bytes/ms (bzw. 1,3 Mbit/s) für BZip2 + Daten-Container auf dem Dokument Hamlet bis 4011 Bytes/ms (bzw. 30,6 Mbit/s) für GZip auf dem Dokument Hamlet. Zu beachten ist, dass – bis auf die Datei Hamlet – Succinct als schnellster Struktur-Kompressor höhere Durchsatzraten erreicht als GZip als schnellster Daten-Kompressor und auch DTD-Subtraktion + DAG als langsamster Struktur-Kompressor erreicht höhere Durchsätze als BZip2 + Daten-Container.

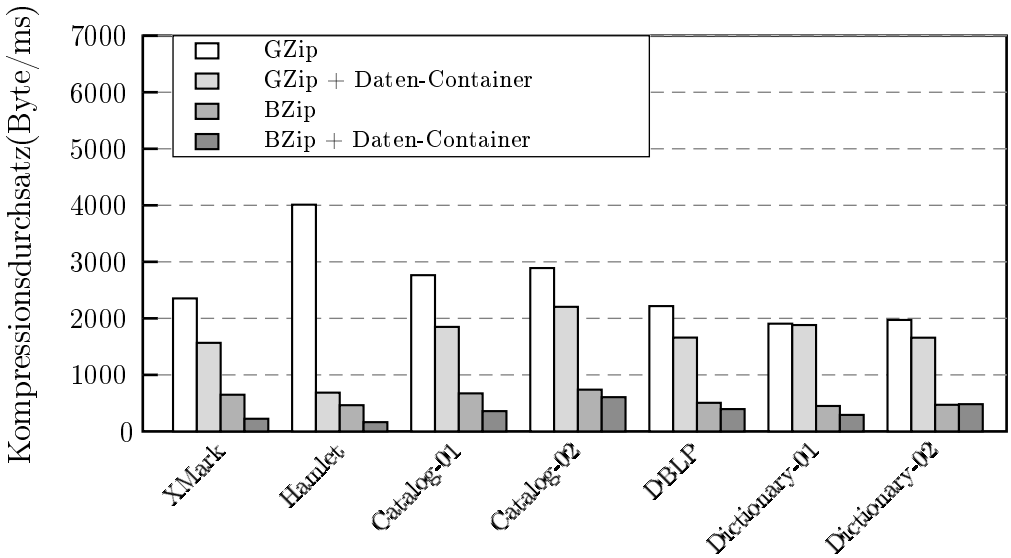

Abbildung 11.9: Vergleich der Durchsätze der Konstanten-Kompressionen

11.2.6 Gesamt-Kompressionszeit

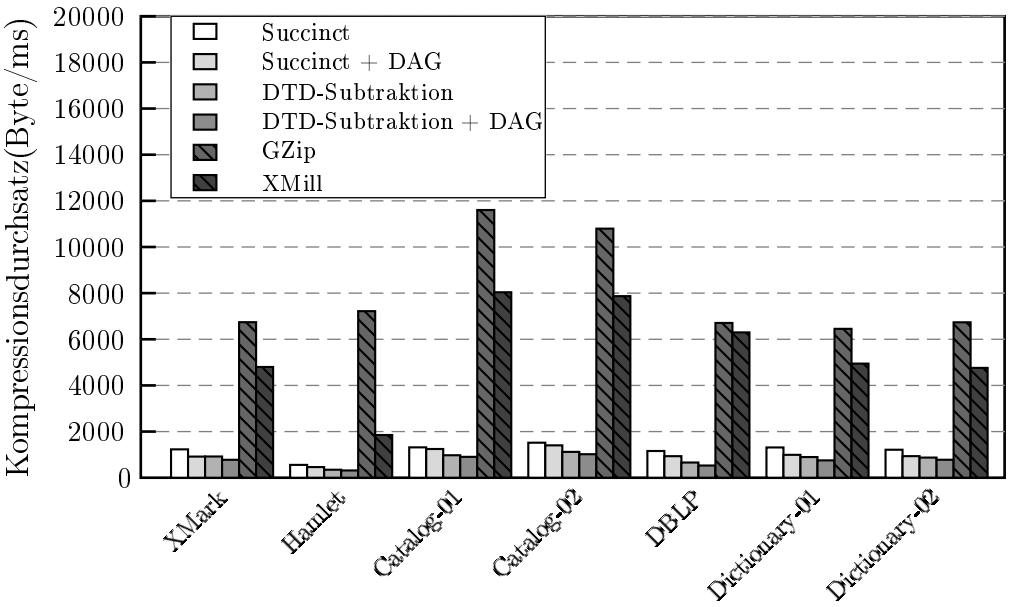

Abbildung 11.10: Vergleich der Durchsätze GZip-basierter Kompressoren

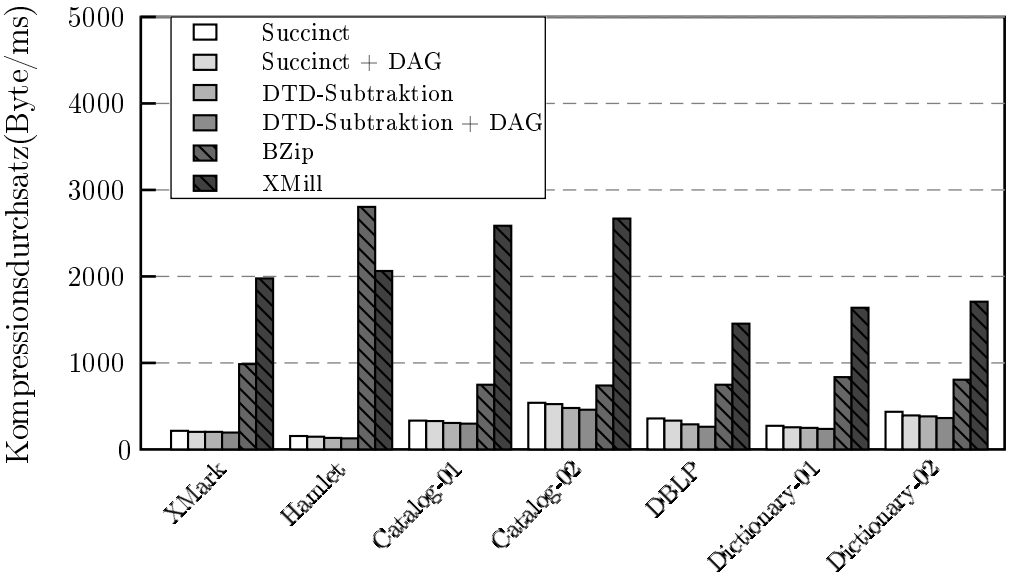

Abbildung 11.11: Vergleich der Durchsätze BZip-basierter Kompressoren

Abbildungen 11.10 und 11.11 zeigen den Vergleich der GZip- bzw. BZip2-basierten Kompressoren hinsichtlich der Kompressionszeit. Hierbei zeigt sich, dass sowohl XMill als auch GZip und BZip2 den vier vorgestellten Verfahren deutlich überlegen sind. Die vorgestellten Verfahren bieten jedoch den Vorteil, dass Anfragen direkt auf dem Komprimat – jedoch ohne vorherige Dekompression – ausgewertet werden können. Auch wird im Allgemeinen nur einmal komprimiert, jedoch werden viele Anfragen auf dem Komprimat auf verschiedenen Empfänger-Rechnern ausgeführt. Daher sind viele Szenarien vorstellbar, bei denen man eine einmalig höhere Kompressionszeit in Kauf nimmt, um vom Vorteil der performanteren Anfrage-Auswertung profitieren zu können.

11.2.7 Dekompressionszeit der Struktur-Kompression

Ein deutlich anderes Bild zeigt sich bei der Dekompressionszeit.

Abbildung 11.12 zeigt die bei der Dekompression der vier vorgestellten Verfahren erreichten Durchsatzraten. Wie auch bei der Kompression erreichen die Succinct-basierten Verfahren höhere Durchsätze als die DTD-Subtraktion-basierten Verfahren. Im Gegensatz dazu sind jedoch die DAG-basierten Varianten schneller als die XML-basierten.

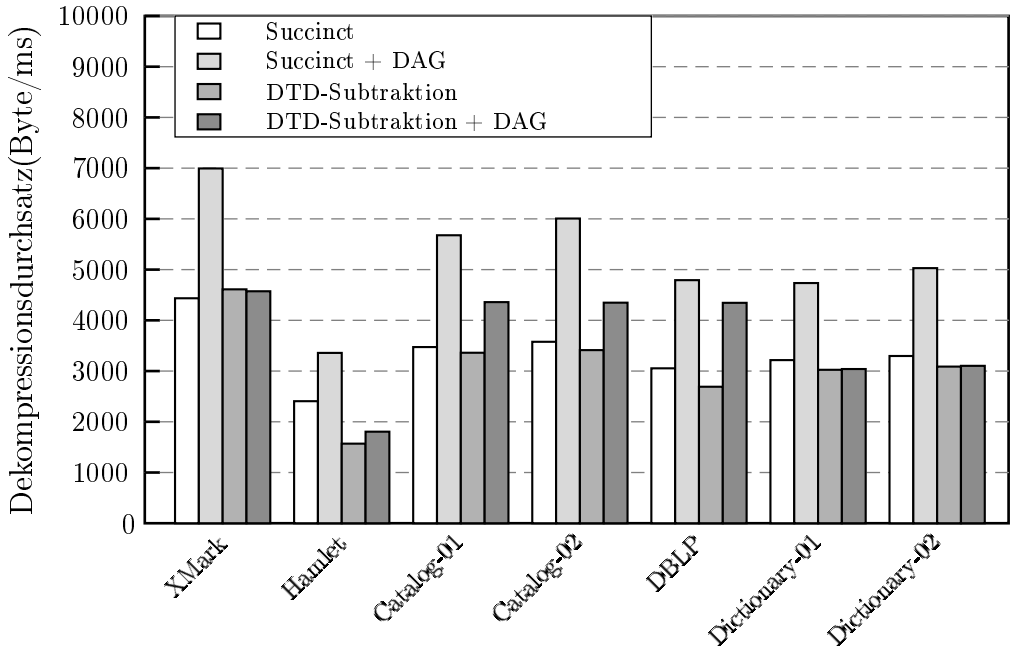

Abbildung 11.12: Vergleich der Dekompressions-Durchsätze der Strukturkompression

Das liegt daran, dass bei der Kompression für die einzelnen Teilbäume die DAG-Kompression und die Succinct- bzw. DTD-Subtraktion-Kompression hintereinander ausgeführt werden müssen, während sie bei der Dekompression stärker miteinander verwoben werden können. Die Dekompression der DAG-basierten Verfahren kann wie bei den XML-basierten Verfahren erfolgen, lediglich, wenn ein DAG-Zeiger erreicht wird, muss der im Arbeitsspeicher gepufferte, wiederholte Teilbaum erneut in die Ausgabe-Datei geschrieben werden. Da dies im Allgemeinen schneller ist, als den entsprechenden Teilbaum erneut zu dekomprimieren, sind bei der Dekompression die DAG-basierten Verfahren schneller als die XML-basierten Verfahren. Da – wie bereits erwähnt – in der Kombination Succinct + DAG mehr Verweise realisiert werden als in der Kombination DTD-Subtraktion + DAG, tritt der Geschwindigkeits-Gewinn bei Succinct + DAG stärker hervor als bei der Kombination DTD-Subtraktion + DAG.

Das schnellste Verfahren – Succinct + DAG – erreicht hierbei Durchsätze von 3358 Bytes/ms (bzw. 25,6 Mbit/s) bis 6994 Bytes/ms (52,4 Mbit/s), es erreicht also höhere Durchsätze als der schnellste ADSL-Standard ADSL2+ mit 25 Mbit/s. Das langsamste Verfahren – DTD-Subtraktion – erreicht Durchsätze von 1569 Bytes/ms (bzw. 12 Mbit/s) bis 4611 Bytes/ms (35,2 Mbit/s).

Hierbei ist der bei der Dekompression erreichte Durchsatz immer höher, als der bei der Kompression erreichte Durchsatz.

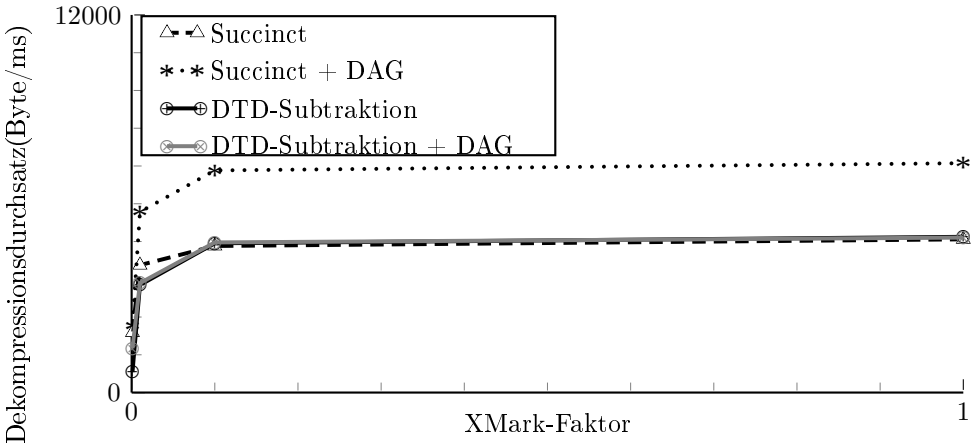

Abbildung 11.13: Skalierung des Dekompressionsdurchsatzes

Bei der Skalierung der Dekompressionsdurchsätze (Abbildung 11.13) zeigt sich wieder, dass nach anfänglichem Anstieg des Durchsatzes bei steigender Dokumentgröße ein nahezu konstanter Durchsatz erreicht wird.

11.2.8 Dekompressionszeit der Konstanten-Kompression

Abbildung 11.14 zeigt die Dekompressions-Durchsätze der Konstanten-Dekompressionen. Auch hier zeigt sich wieder, dass die GZip-basierten Dekompressoren schneller dekomprimieren als die BZip2-basierten Dekompressoren. Hinsichtlich der Durchsätze der Container-basierten Varianten im Vergleich zu den Container-losen Varianten kann jedoch keine klare Aussage getroffen werden. Vermutlich ist dies darin begründet, dass die Container deutlich weniger Elemente enthalten als die komprimierten Datenmengen in den Container-losen Varianten, so dass in den Container-losen Varianten in vielen Fällen performanter dekomprimiert werden kann.

Während die BZip2-basierten Konstanten-Kompressoren deutlich geringere Durchsätze erreichen als die Struktur-Kompressoren, erreichen die GZip-basierten Konstanten-Kompressoren vergleichbare Durchsätze wie die Struktur-Kompressoren. Die Durchsätze reichen von 424 Bytes/ms (bzw. 3,2 Mbit/s) für BZip2 + Daten-Container auf Hamlet bis zu 7275 Bytes/ms (bzw. 55,5 Mbit/s) für GZip + Daten-Container auf Catalog-02.

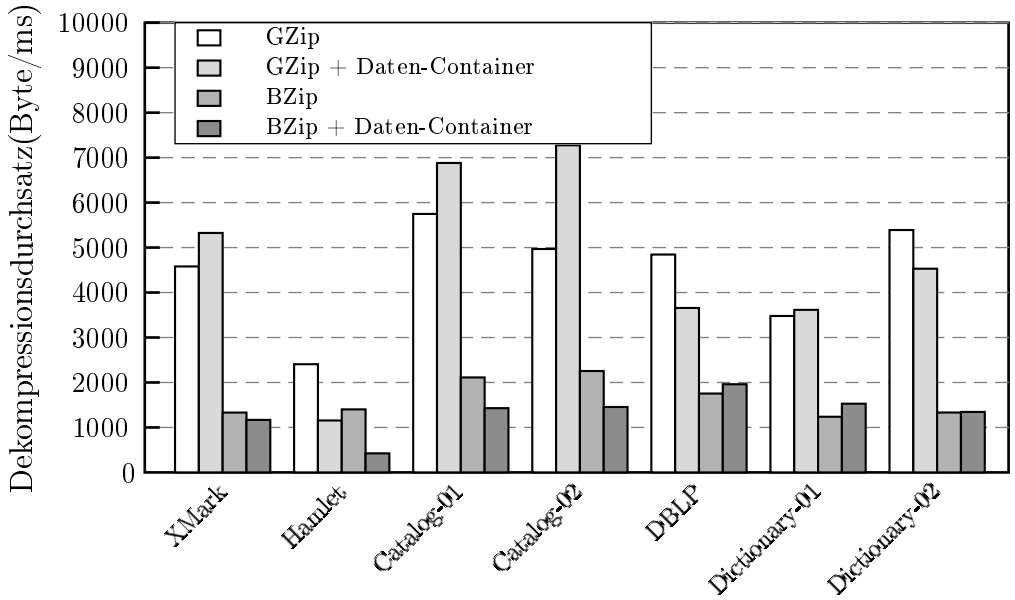

Abbildung 11.14: Vergleich der Durchsätze der Konstanten-Dekompressionen

11.2.9 Gesamt-Dekompressionszeit

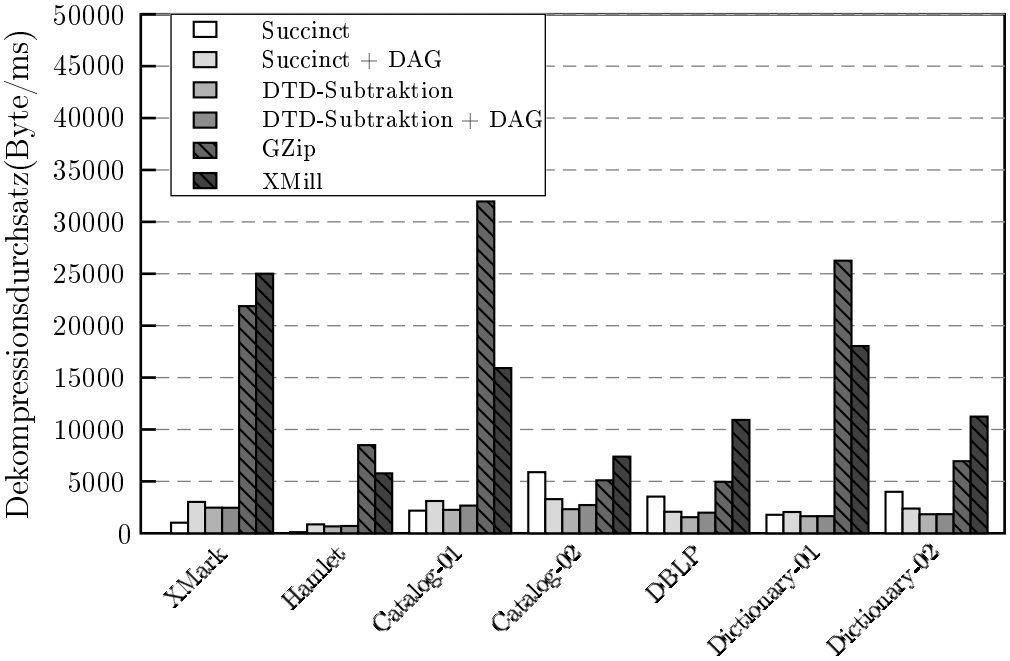

Abbildung 11.15: Vergleich der Durchsätze GZip-basierter Dekompressoren

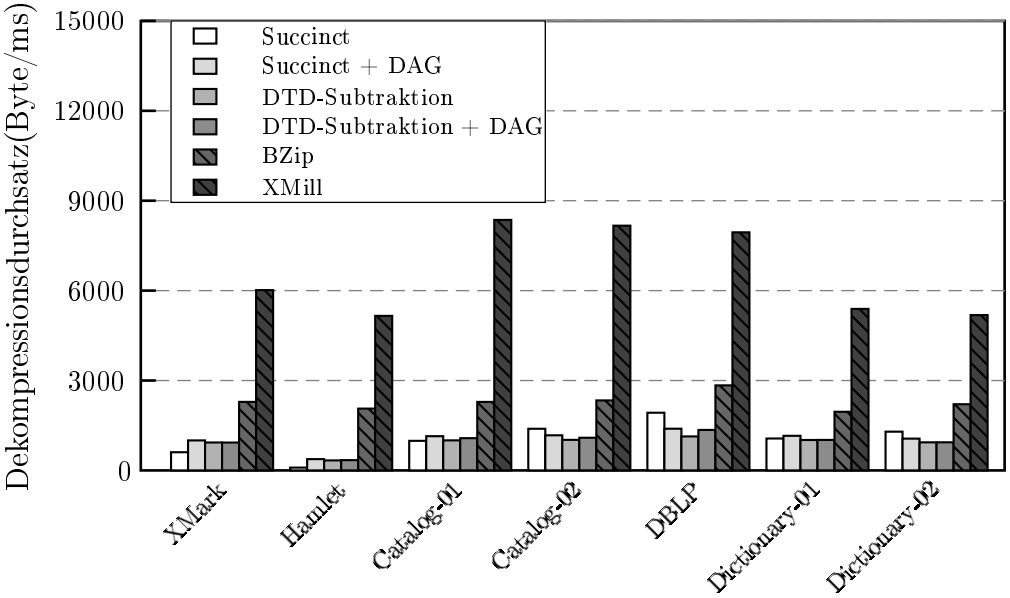

Abbildung 11.16: Vergleich der Durchsätze BZip-basierter Dekompressoren

Tabelle 11.1: Anfragen des Benchmarks XPathMark-A

ID	Query
Q1	/site/closed_auctions/closed_auction/annotation/description/text/keyword
Q2	//closed_auction//keyword
Q3	/site/closed_auctions/closed_auction//keyword
Q4	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date
Q5	/site/closed_auctions/closed_auction[descendant::keyword]/date
Q6	/site/people/person[profile/gender and profile/age]/name
Q7	/site/people/person[phone or homepage]/name
Q8	/site/people/person[address and (phone or homepage) and (creditcard or profile)]/name

Im Vergleich zum Gesamt-Dekompressions-Durchsatz der Kompressoren XMill, GZip und BZip2 (vergleiche Abbildungen 11.15 und 11.16) zeigt sich wiederum, dass die Durchsätze von XMill, GZip und BZip2 die Durchsätze der in dieser Arbeit vorgestellten Verfahren deutlich übersteigen. Da jedoch aufgrund der Abfragbarkeit oftmals auf eine vollständige Dekompression verzichtet werden kann, stellt dies nicht unbedingt einen Nachteil der in dieser Arbeit vorgestellten Verfahren dar.

11.3 Auswertungszeit

Zur Evaluierung der Anfrage-Auswertungszeiten wurde ein XPath-Framework für Datenströme entsprechend Kapitel 9 genutzt.

Es wurden die Basis-Verfahren Succinct, DTD-Subtraktion und DAG verglichen mit den folgenden beiden Verfahren:

- Uncompressed – Das XPath-Framework wurde wie Kapitel 9 beschrieben auf einem unkomprimierten SAX-Event-Strom angewandt.
- JAXP (Java API for XML Processing) – die in Java enthaltene Standard-API zum Validieren und Parsen von XML-Dokumenten.

Als Test-Dokumente wurden 8 XML-Dokumente durch den XMark-Benchmark generiert mit Skalierungsfaktoren 0,001, 0,002, 0,004, 0,008, 0,016, 0,032, 0,064 und 0,128. Ausgewertet wurden auf diesen die Anfragen des XPath-Benchmarks XPathMark-A [45], welche als Anfragen Q1, ..., Q8 in Tabelle 11.1 gelistet sind.

Für alle getesteten Verfahren wurde die Dauer der Ermittlung der Ergebnis-Knoten der XPath-Anfrage gemessen. Dabei wurden nicht die darunter liegen-

den Teilbäume ausgegeben, da es von der weiteren Anwendung abhängt, ob man z.B. die Teilbäume komprimiert oder dekomprimiert erhalten möchte.

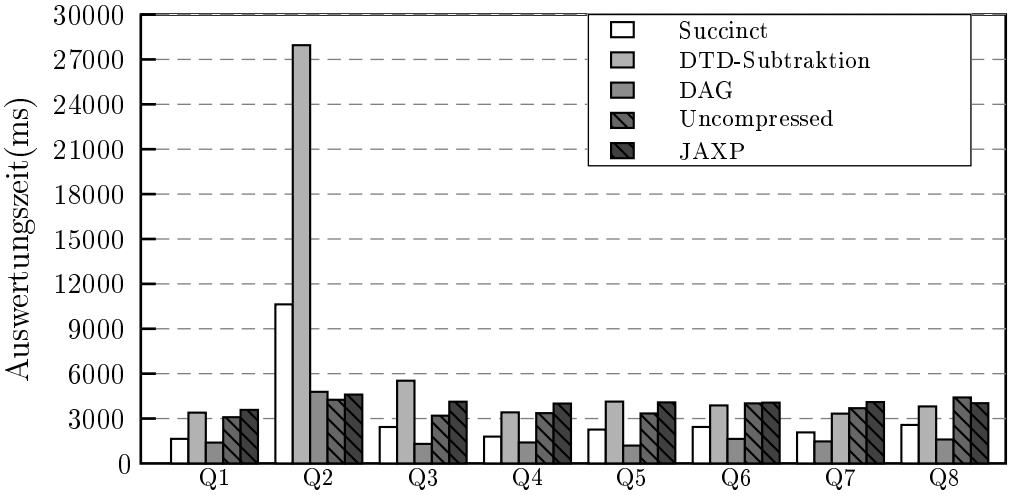

Abbildung 11.17: Vergleich der Anfrage-Auswertungszeiten

Abbildung 11.17 gibt zunächst einen Gesamtüberblick über alle Anfragen angewandt auf das Dokument mit Skalierungsfaktor 0,128. Bei den Anfragen Q1, Q4, Q6, Q7 und Q8 zeigt sich, dass der DAG die schnellste Auswertungszeit vorweisen kann, gefolgt von der Succinct-Darstellung, gefolgt von DTD-Subtraktion und schließlich gefolgt von JAXP und Uncompressed. Bei den Anfragen Q3 und Q5 ist die Auswertungszeit für DTD-Subtraktion höher als für Uncompressed und JAXP, während sie für DAG und Succinct noch niedriger ist. Bei Anfrage Q2 schließlich ist die Auswertungszeit für DAG, Succinct und DTD-Subtraktion höher als für JAXP und Uncompressed.

Vergleichen wir diese Anfragen, so sehen wir, dass Q2, Q3 und Q5 jeweils mindestens eine descendant-Achse enthalten, während die restlichen Anfragen nur aus child-Achsen bestehen. Beginnt eine Anfrage mit einem descendant-Achsen-Schritt (wie z.B. Q2), so bedeutet dies, dass jeder Knoten des Dokumentes ein potentieller Treffer ist, es kann also bei der Auswertung kein Knoten übersprungen werden. Je später jedoch ein descendant-Achsen-Schritt vorkommt, desto mehr Teile können übersprungen werden und desto größer ist der Vorteil der komprimierten Repräsentationen gegenüber dem unkomprimierten XML.

Der DAG erhält hierbei einen noch größeren Vorteil als die anderen Verfahren, da dieser bereits – wie bei DOM – einen direkten Zeiger auf first-child und next-sibling enthält, während bei der Succinct-Darstellung der Bit-Strom und bei der DTD-Subtraktion DTD und KST bis zum gewünschten Knoten geparst werden müssen.

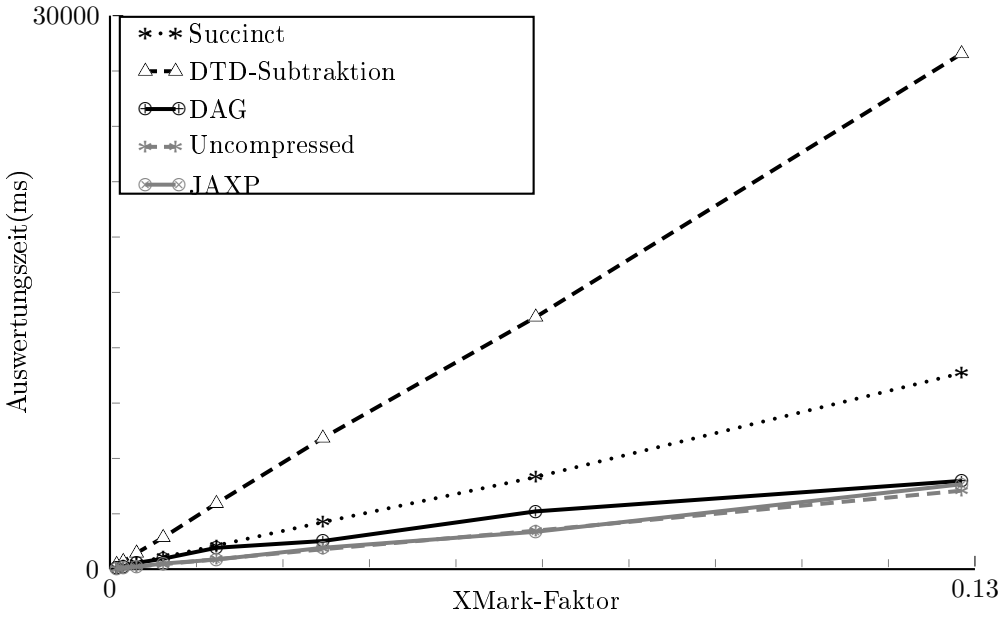

Abbildung 11.18: Anfrage-Auswertungszeit für Anfrage Q2

Abbildungen 11.18, 11.19 und 11.20 zeigen die Skalierung der Auswertungszeiten für die Anfragen Q2 (descendant-Achse zu Beginn), Q5 (descendant-Achse in Tiefe 4 + Prädikatfilter) und Q7 (nur child-Achsen, Prädikatfilter mit Disjunktion).

Wie zu sehen ist, skalieren alle Verfahren bei steigender Dokumentgröße nahezu linear, wobei die Rangfolge der Verfahren (wie in Abbildung 11.17 zu sehen) von der Position der ersten descendant-Achse abhängt.

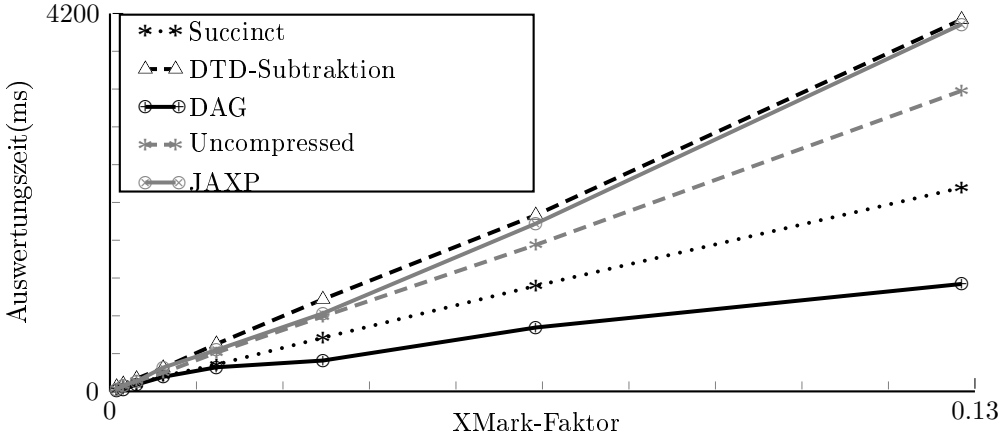

Abbildung 11.19: Anfrage-Auswertungszeit für Anfrage Q5

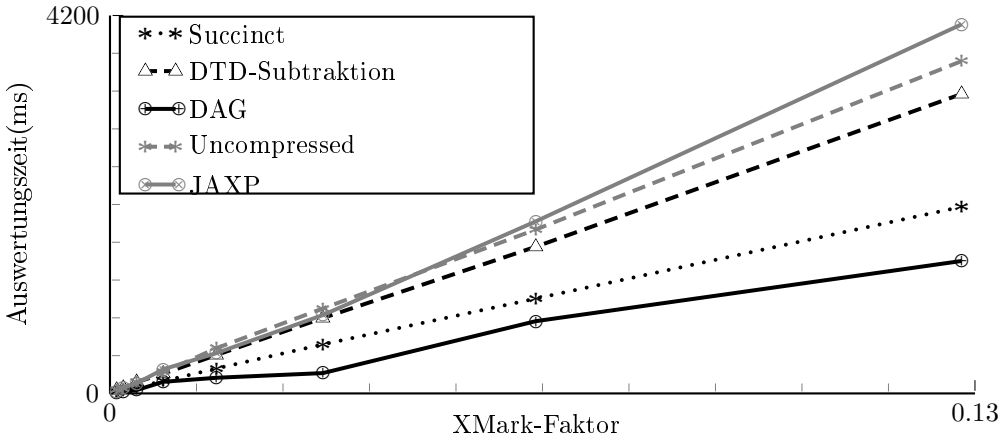

Abbildung 11.20: Anfrage-Auswertungszeit für Anfrage Q7

Geht man nicht vom ungünstigsten Fall – also einer Anfrage beginnend mit descendant-Achse – aus, so ist die Anfrage-Auswertung auf den Komprimaten schneller oder mindestens genauso schnell wie auf unkomprimiertem XML und damit auf jeden Fall schneller, als wenn man einen nicht-navigierbaren Kompressor (z.B. XMill, GZip oder BZip2) verwenden würde und vor der Anfrage-Auswertung dekomprimieren würde.

11.4 Fazit

Als Ergebnis dieser Messreihe kann man sehen, dass keines der vorgestellten Basis-Verfahren den jeweils anderen beiden in allen gemessenen Aspekten überlegen ist: während DTD-Subtraktion die stärkste Kompression erreicht, erreicht die Succinct-Darstellung die größten Kompressions- und Dekompressions-Durchsätze und der DAG erreicht die schnellste Auswertungszeit.

Je nach Anwendung und den daraus resultierenden Anforderungen an diese Eigenschaften, kann man somit ein geeignetes Verfahren auswählen.

Auch bei der Konstanten-Kompression zeigt sich ein deutlicher Trade-Off zwischen Kompressionsrate und Kompressionszeit. Da jedoch diese Verfahren komplett unabhängig von der Struktur-Kompression sind, kann auch hier ein Verfahren mit den gewünschten Eigenschaften ausgewählt werden und beliebig mit einer geeigneten Struktur-Kompression kombiniert werden.

Insgesamt hat sich im Vergleich mit anderen Verfahren gezeigt, dass insbesondere hinsichtlich der jeweiligen Stärke des jeweiligen Verfahrens die in dieser Arbeit entwickelten und vorgestellten Verfahren die anderen getesteten Verfahren übertreffen: einerseits hinsichtlich der Kompressionsstärke (z.B.

DTD-Subtraktion + DAG) andererseits hinsichtlich der Kompressionszeit (z.B. Succinct-Verfahren) oder hinsichtlich Anfrage-Auswertungszeit (z.B. DAG).

Die Kombination der Verfahren Succinct mit DAG und DTD-Subtraktion mit DAG erreicht erstaunlicherweise nicht nur eine stärkere Kompression als diese Verfahren mit XML sondern auch eine schnellere Dekompression. Lediglich bei der Kompressionszeit muss man den Trade-Off eingehen und erhält eine etwas höhere Kompressionszeit.

12 Anwendungen für komprimierte XML-Repräsentationen

In diesem Kapitel werde ich die in Kapitel 1 vorgestellten Anwendungen wieder aufgreifen und erörtern, von welchen der in dieser Arbeit vorgestellten Kompressionsverfahren sie aufgrund der Gewichtung der Anforderungen besonders stark profitieren.

12.1 News-Ticker

Das News-Ticker-Szenario besteht aus einer Datenquelle – z.B. einer Nachrichten-Agentur – die einen kontinuierlichen Strom an Nachrichten produziert, einem Bezieher, der nur an einem Teil der produzierten Nachrichten interessiert ist, und einem Nachrichten-Broker, der die Interessen des Beziehers kennt, die für ihn interessanten Nachrichten aus dem Datenstrom herausfiltert und an den Bezieher weiterleitet.

In diesem Szenario sind insbesondere die Anforderungen 1-9 aus Kapitel 1.3 wichtig, es muss also eine starke Kompression erreicht werden, Kompressions- und Dekompressionsdurchsatz dürfen nicht unter dem Datendurchsatz der Datenquelle liegen, und atomare Anfrage-Auswertung muss unterstützt werden, damit der Nachrichten-Broker effizient die Nachrichten filtern kann. Anforderung 10 – die Unterstützung von Updates – ist in diesem Beispiel nicht relevant, da keinerlei Updates auf dem Nachrichtenstrom vorgesehen sind.

In Anbetracht dieser Anforderungen empfiehlt sich daher für dieses Szenario die Verwendung des Kompressions-Verfahrens DTD-Subtraktion, welches eine starke Kompression bei Kompressionsdurchsätzen von ca. 8 Mbit/s erreicht und auch die Anfrage-Auswertung einfacher Pfad-Anfragen effizient unterstützt.

Als weitere Alternative empfiehlt sich auch, das Succinct-Verfahren einzusetzen. In diesem Fall erhielte man einen erhöhten Kompressionsdurchsatz und

eine etwas effizientere Anfrage-Auswertung beim Nachrichten-Broker, im Gegenzug erhielte man allerdings eine etwas verringerte Kompressionsstärke.

Lediglich die DAG-basierten Varianten Succinct + DAG und DTD-Subtraktion + DAG empfehlen sich aufgrund der verringerten Kompressions-Durchsätze nur bedingt für dieses Szenario.

12.2 Daten-Management für mobile, Ajax-basierte Web 2.0 Anwendungen

Das Ajax-Szenario besteht aus einem XML-Server und einem XML-Client in Form eines DOM-basierten Web-Browsers. Der Server enthält das gesamte Dokument, während der Client nur wenige Ausschnitte des Dokumentes enthält. Sobald der Client weitere Informationen vom Server benötigt, werden diese asynchron in Form eines XML-Fragmentes vom Server zum Client gesendet und mittels JavaScript in den DOM-Baum des Clients zur Laufzeit integriert, so dass dem Benutzer des Clients eine interaktive Anwendung suggeriert wird.

Ersetzt man die DOM-Komponente auf Client-Seite durch eine Navigations- und Update-fähige, komprimierte XML-Repräsentation, so können die übrigen Ajax-Komponenten unverändert übernommen werden. Statt unkomprimiertem XML wird komprimiertes XML übertragen, so dass Übertragungskosten eingespart werden. Die Darstellung der komprimierten XML-Repräsentation erfordert im Hauptspeicher deutlich weniger Speicher als die Darstellung des eigentlichen DOM-Baumes bei gleichem Funktionsumfang, so dass Arbeitsspeicher eingespart werden kann. Dadurch können bei gleichem Arbeitsspeicher deutlich umfangreichere Ajax-Anwendungen umgesetzt werden, und somit wird es auch mobilen Kleinstgeräten (wie z.B. Mobiltelefonen und PDAs) ermöglicht, Ajax-basierte Web 2.0 Anwendungen zu nutzen.

In diesem Szenario sind besonders die Anforderungen 1-2 sowie 6-10 wichtig, also eine starke Kompression sowie Auswertung von Anfragen und Updates direkt auf dem Komprimat. Die Kompressions- und Dekompressionsdurchsätze (Anforderungen 4-5) spielen eine eher untergeordnete Rolle, da kein kontinuierlicher Datenstrom sondern nur kleine XML-Fragmente versendet werden.

Aufgrund der guten Kompressionsstärke bei sehr effizienter Unterstützung der DOM-Funktionalitäten, empfiehlt sich zur Verbesserung von Ajax insbesondere die Kombination aus DAG und Succinct-Darstellung als Kompressionsmethode. Da aber alle in dieser Arbeit vorgestellten Kompressionsverfahren die DOM-Funktionalitäten unterstützen, kann prinzipiell auch eines der anderen vorgestellten Verfahren benutzt werden.

Teile der Ideen eines durch XML-Kompression verbesserten Daten-Managements für mobile, Ajax-basierte Web 2.0 Anwendungen wurden in [21] veröffentlicht.

12.3 Verbesserung der Cache-Kapazität durch Kompression

Das Cache-Szenario umfasst einen Server, der das komplette, komprimierte XML-Dokument enthält, sowie einen client-seitigen Cache, der die komplette, komprimierte Dokument-Struktur enthält zuzüglich einiger weniger, komprimierter Text-Knoten, sofern diese für die Beantwortung früherer Anfragen benötigt wurden.

Diese Architektur erlaubt dem Client einerseits, effizient zu entscheiden, ob er zur Beantwortung einer neuen Anfrage bereits alle Daten im Cache hat, andererseits entfällt in diesem Szenario auch der Overhead aufgrund von Knoten-IDs, wie sie in anderen XML-basierten Caching-Szenarien notwendig sind, da Reihenfolge der Anfrage-Auswertung sowie Dokument-Struktur genügen, um für jeden Text-Knoten dessen Position im Struktur-Komprimat eindeutig zu bestimmen.

In diesem Szenario sind insbesondere die Anforderungen 1-2 sowie 6-10 wichtig, also eine starke Kompression sowie Auswertung von Anfragen und Updates direkt auf dem Komprimat. Die Kompressions- und Dekompressionsdurchsätze (Anforderungen 4-5) spielen eine eher untergeordnete Rolle, haben aber eine deutlich stärkere Bedeutung als im vorangehenden Szenario. Dadurch, dass die komplette, komprimierte Struktur im Cache vorhanden ist, erhalten die Anforderungen 1-2 ein deutlich höheres Gewicht als die übrigen Anforderungen.

Für dieses Szenario eignet sich daher insbesondere das Kompressions-Verfahren DTD-Subtraktion, da es eine sehr kleine Struktur-Repräsentation erzeugt. Ebenso unterstützt dieses Verfahren das in Kapitel 9 vorgestellte XPath-Auswertungs-Verfahren, welches die Dokument-Reihenfolge der Text-Knoten garantiert. Prinzipiell können aber alle im Rahmen dieser Arbeit vorgestellten Kompressions-Verfahren zur Umsetzung eines solchen Szenarios eingesetzt werden.

Zusammengefasst bietet der Einsatz solch eines neuen Caching-Verfahrens die folgenden Vorteile:

- Aufgrund der Kompression erhöht sich die Cache-Kapazität, dadurch werden mehr Cache-Hits ermöglicht.
- Weniger Datentransfer durch komprimierte Übertragung.

- Kein zusätzlicher Overhead durch Identifizierungs-Informationen für die Zuordnung von Blattknoten der Struktur zu Konstanten.
- Der Client kann Vollständigkeit der Daten in seinem Cache bzgl. einer Anfrage entscheiden.

Teile der Ideen eines durch Struktur-Kompression verbesserten Cachings wurden in [14] zur Veröffentlichung eingereicht.

13 Schlussbetrachtungen

13.1 Zusammenfassung

In dieser Arbeit wurden drei verschiedene Verfahren – Succinct-Darstellung, DAG-basierte Kompression und DTD-Subtraktion – zur navigierbaren Kompression der Struktur von XML-Datenströmen vorgestellt. Jedes dieser Verfahren besitzt die Eigenschaft, dass sowohl Anfragen als auch Updates direkt auf dem Komprimat – ohne vorherige Dekompression und anschließende Kompression – durchgeführt werden können.

Für jedes der drei Verfahren wurde bewiesen, dass Kompression und Dekompression Umkehroperationen zueinander sind, dass also die Hintereinanderausführung von Kompression und Dekompression auf einem XML-Struktur-Strom xml wieder den ursprünglichen Struktur-Strom xml herstellt. Ebenso wurde für jedes dieser Verfahren die Korrektheit der Basis-Navigation basierend auf den atomaren XPath-Achsen first-child, next-sibling und parent sowie auf den Funktionen getLabel und getType nachgewiesen. Die Möglichkeit der Abbildung der kompletten DOM-Schnittstelle direkt auf dem Komprimat mit Hilfe von Basis-Navigation und Updates wurde ebenfalls für alle drei Verfahren nachgewiesen.

Neben den drei grundlegenden Verfahren wurden auch zwei hybride Verfahren – Succinct + DAG und DTD-Subtraktion + DAG – vorgestellt, die jeweils eine Kombination zweier grundlegender Verfahren darstellen.

Für die zuvor von der XML-Struktur getrennten Konstanten wurde eine Reihe von verschiedenen, bereits existierenden Kompressions-Verfahren vorgestellt und deren Eigenschaften erörtert. Ebenso wurden die Vor- und Nachteile verschiedener Integrationsmöglichkeiten von komprimierter Struktur und komprimierten Daten diskutiert.

Um nicht nur eine Basis-Navigation zu unterstützen, sondern den weit verbreiteten XPath-Standard, wurde ein Verfahren zur XPath-Anfrage-Auswertung vorgestellt. Dieses Anfrage-Auswertungs-Verfahren kann auf jeder XML-Repräsentation – egal ob komprimiert oder nicht komprimiert – die die Basis-

Navigation, bestehend aus den Achsen first-child, next-sibling und parent sowie aus den Funktionen getLabel und getType, unterstützt, mehrere XPath-Anfragen parallel in einer zur Dokument-Größe proportionalen Zeit auswerten.

Abschließend wurden in einer Reihe von Messungen die drei grundlegenden Verfahren und die beiden hybriden Varianten sowohl untereinander als auch mit anderen frei verfügbaren XML- bzw. Daten-Kompressoren verglichen und bzgl. Kompressionsstärke, Kompressions- und Dekompressionszeit sowie Anfrage-Auswertungszeit evaluiert. Obwohl die vorgestellten Verfahren im Gegensatz zu den anderen getesteten Verfahren navigierbar und Update-fähig sind, erreichen sie dennoch vergleichbare Kompressionsstärken, wenn auch die Kompressions- und Dekompressionszeiten etwas größer sind als die der reinen Kompressoren. Ebenso zeigte sich, dass die Anfrage-Auswertungszeiten durchaus vergleichbar und in vielen Fällen sogar besser sind, als diejenigen von XPath-Auswertern auf unkomprimiertem XML.

Im Vergleich der Verfahren untereinander hat sich gezeigt, dass keines der Verfahren den jeweils anderen absolut überlegen ist. Jedes der Verfahren hat seine Stärke in einem anderen Bereich: Während die DTD-Subtraktion besonders stark komprimiert, sind die Kompressions- und Dekompressionszeiten der Succinct-Darstellung besonders niedrig und die Anfrage-Auswertung auf der DAG-Kompression besonders effizient. Je nach Anwendung und den daraus resultierenden Anforderungen kann entsprechend eines der vorgestellten Verfahren in Kombination mit einem für die Anwendung geeigneten, vom Struktur-Kompressions-Verfahren unabhängigen Daten-Kompressions-Verfahren gewählt werden.

13.2 Erfüllung der Anforderungen

Zusammenfassend werde ich nun noch einmal die in Kapitel 1 vorgestellten Anforderungen aufgreifen und erörtern, in welchem Maße diese durch die in dieser Arbeit vorgestellten Lösungen erfüllt werden.

- **Anforderung 1:** *Kompression und Dekompression müssen zueinander invers sein, die Dekompression der komprimierten Repräsentation muss also bei Eingabe eines beliebigen validen Dokuments wieder das ursprüngliche Dokument erzeugen.*
  Die Erfüllung dieser Anforderung wird für die Succinct-Darstellung in den Sätzen 4.4 und 4.8 nachgewiesen. Für die DAG-basierte Kompression wird die Erfüllung dieser Anforderung in Satz 5.1 und für die DTD-Subtraktion in Satz 6.1 nachgewiesen.

- **Anforderung 2:** *Die Kompressionsrate muss mindestens so stark sein wie die anderer XML-Kompressions-Verfahren mit vergleichbaren Eigenschaften.*
  Wie in Abbildungen 11.5 und 11.6 zu sehen, erreichen alle getesteten Verfahren Kompressionsraten vergleichbar zu XMill, BZip2 und GZip, obwohl keines der letzteren Verfahren Navigation direkt auf dem Komprimat unterstützt. Somit kann keines dieser Verfahren vergleichbare Eigenschaften vorweisen.
- **Anforderung 3:** *Kompression und Dekompression müssen vergleichbare Durchsätze erreichen wie derzeit übliche Übertragungsverfahren.*
  Wie in Abbildung 11.7 zu sehen ist, erreichen die gemessenen Verfahren Durchsätze von 4,4Mbit/s bis zu 53,5Mbit/s. Im Vergleich zum derzeit üblichen Übertragungsverfahren ADSL mit Durchsätzen von unter 1Mbit/s bis zu 8Mbit/s werden also durchaus vergleichbare Durchsätze erreicht.
- **Anforderung 4:** *Die Dekompression muss mindestens so schnell sein wie die Kompression.*
  Im Vergleich der Abbildungen 11.7 und 11.12 ist zu sehen, dass für jedes der gemessenen Verfahren die Dekompression höhere Durchsätze erreicht als die Kompression.
- **Anforderung 5:** *Kompression und Dekompression müssen möglich sein, ohne dass das gesamte Dokument beziehungsweise das gesamte Komprimat bekannt ist.*
  In den Kapiteln 4.3, 5.2 und 6.2 wird die fenster-basierte Kompression von unendlichen Datenströmen und somit die Erfüllung dieser Anforderung erläutert.
- **Anforderung 6:** *Zu jedem Knoten des ursprünglichen XML-Dokumentes muss eine eindeutige Repräsentation im Komprimat existieren.*
  Für die Succinct-Darstellung genügt eine Position p im Bitstrom, für den DAG ein Stack bestehend aus Paaren (ID, Knotentyp), welcher den Pfad zum aktuellen Knoten repräsentiert, und für die DTD-Subtraktion genügt ein Tupel (KST, n, p) aus Komprimat KST, Syntaxknoten n und Position p im KST, um einen Knoten des ursprünglichen XML-Dokumentes eindeutig zu identifizieren.
- **Anforderung 7:** *Partielle Dekompression, also Dekompression von XML-Teilbäumen innerhalb des Komprimats, muss möglich sein.*
  Da die Sätze 4.4, 4.8, 5.1 und 6.1 und deren Beweise, die die Korrektheit von Kompression und Dekompression nachweisen, nicht nur auf das komplette Dokument, sondern auch auf Teilbäume bzw. deren Entsprechung im Struktur-Strom anwendbar sind, ist diese Anforderung erfüllt.

- **Anforderung 8:** *Die Basis-Operationen first-child, next-sibling, parent sowie die Ermittlung des Typs und des Labels eines Knotens direkt auf dem Komprimat müssen unterstützt werden.*
  Wie in den Kapiteln 4.1.2, 4.4.1, 5.3, 5.4.1, 6.3 und 6.4.1 gezeigt, werden von allen drei Verfahren die Basis-Navigation und somit diese Anforderung erfüllt.
- **Anforderung 9:** *Die Anfrage-Auswertungszeiten auf dem Komprimat sollten hierbei vergleichbar zu Anfrage-Auswertungszeiten auf unkomprimiertem XML sein.*
  Wie z.B. in Abbildung 11.17 zu sehen ist, erreichen alle drei Verfahren vergleichbare bzw. sogar teilweise bessere Anfrage-Auswertungszeiten als die der Standard-XPath-Auswertern auf unkomprimiertem XML.
- **Anforderung 10:** *Die Basis-Operationen insert und remove müssen direkt auf dem Komprimat unterstützt werden.*
  Wie in den Kapiteln 4.4.3, 5.4.2 und 6.4.2 erörtert, unterstützen alle drei Verfahren diese Basis-Operationen direkt auf dem Komprimat.

13.3 Ausblick

In diesem Kapitel werde ich Ideen zur zukünftigen Erweiterung und Verbesserung der in dieser Arbeit vorgestellten Ansätze vorstellen.

13.3.1 Verbesserte Konstanten-Kompression

Wie sich in den Messungen gezeigt hat, wird die Struktur des XML-Dokumentes sehr stark komprimiert, so dass die komprimierte Struktur nur noch 0,3% bis 16,7% der ursprünglichen Dokumentgröße beträgt. Im Vergleich dazu komprimiert die Konstanten-Kompression mit 9,2% bis 30,4% deutlich schwächer.

Es ist daher zu vermuten, dass die Entwicklung eines geeigneten Daten-Kompressors noch einmal einen deutlichen Fortschritt in der XML-Kompression bringen würde.

Da jedoch die Text-Kompression als Forschungsgebiet deutlich länger existiert als die XML-Kompression, ist zu erwarten, dass die derzeit verfügbaren Text-Kompressoren bereits sehr leistungsstark sind. Daher sollte der Fokus der zukünftigen Forschung nicht unbedingt auf der Entwicklung verbesserter Text-Kompressoren liegen, sondern eher darauf, wie man mit Hilfe der semantischen Informationen, die durch die XML-Struktur gegeben sind, die vorhandenen Text-Kompressoren so erweitern kann, dass sie eine verbesserte Kompression erreichen können.

Einen Einstieg in diese Forschung stellt z.B. das Sortieren der Text-Konstanten in semantische Container dar, wie es schon von XMill vollzogen wurde.

13.3.2 Unterstützung aller XML-Anwendungen

Mit der Unterstützung von XPath-Pfad-Anfragen und der Unterstützung von einfachen Update-Operationen wurde in dieser Arbeit ein Anfang gemacht, alle Anwendungen, die auf XML möglich sind, auch auf der entsprechenden komprimierten Repräsentation durchführen zu können. XPath stellt dabei zwar einen grundlegenden Baustein dar, jedoch existieren eine Vielzahl weiterer Anwendungen und Standards, die teilweise auf den Grundlagen von XPath beruhen. Beispiele für solche Standards sind z.B. die XML-Anfragesprache XQuery und die Programmiersprache zur Transformation von XML-Dokumenten XSLT. Beide bieten eine Vielzahl von Operationen, die über die Basis-Navigation via XPath hinausgehen.

Um also komprimiertes XML in gleichem Maße nutzbar zu machen wie herkömmliches XML, müsste man die hier vorgestellten Verfahren untersuchen, inwieweit diese in der Lage sind, weitere XML-basierte Standards zu unterstützen, bzw. inwieweit diese Verfahren anpassbar sind, so dass eine Unterstützung der Standards gewährleistet werden kann.

Auch bei der schema-basierten Kompression – DTD-Subtraktion – habe ich mich in dieser Arbeit auf die Unterstützung einer Schema-Sprache – DTD – beschränkt. Neben DTD existieren aber noch andere Schema-Sprachen, wie z.B. XML Schema oder RelaxNG, welche eine höhere Mächtigkeit als die DTD besitzen. Hier bliebe es also zu untersuchen, ob die in dieser Arbeit vorgestellten Ergebnisse zur schema-basierten Kompression auf andere Schema-Sprachen übertragbar sind, ob der höhere Sprachumfang eine Übertragung der Ergebnisse nicht möglich macht, oder ob der höhere Sprachumfang zu schwächeren oder stärkeren Kompressions-Ergebnissen führt.

13.3.3 Verbesserte Navigation durch Indizierung

Obwohl die in Kapitel 1 dieser Arbeit vorgestellten Anwendungen auch die Unterstützung von Navigation und teilweise auch von Updates direkt auf dem Komprimat erforderten, lag dennoch der Hauptfokus der Anwendungen auf einer starken Kompression.

Möchte man nun statt dieser relativ einfachen Anwendungen ein komplettes natives XML-Datenbank-System basierend auf komprimiertem XML aufbauen, so erhalten Navigation und Updates einen deutlich höheren Stellenwert.

Für solch ein natives XML-Datenbank-System wäre es also interessant, die komprimierten XML-Repräsentationen um zusätzliche Index-Informationen anzureichern (z.B. direkte Pointer zu next-siblings), die eine optimierte Anfrage-Auswertung erlauben. Dabei ist natürlich auch die Kompression nicht aus dem Blickfeld zu verlieren: Ziel sollten Kompressions-Verfahren sein, die dennoch eine starke Kompression (wenn auch eine etwas schwächere Kompression als die

in dieser Arbeit vorgestellten Verfahren) erreichen, aber die mit Hilfe von Indizes deutliche Performanz-Steigerungen bei der Anfrage-Auswertung und bei der Durchführung von Updates direkt auf der komprimierten Repräsentation erreichen.

Literaturverzeichnis

[1] *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*. IEEE Computer Society, 2002.

[2] *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*. Morgan Kaufmann, 2002.

[3] *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*. IEEE Computer Society, 2004.

[4] *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. IEEE Computer Society, 2005.

[5] Joaquín Adiego, Gonzalo Navarro, and Pablo de la Fuente. Lempel-Ziv Compression of Structured Text. In *2004 Data Compression Conference (DCC 2004), 23-25 March 2004, Snowbird, UT, USA*, pages 112–121. IEEE Computer Society, 2004.

[6] Gennady Antoshenkov. Dictionary-Based Order-Preserving String Compression. *VLDB J.*, 6(1):26–39, 1997.

[7] Gennady Antoshenkov, David B. Lomet, and James Murray. Order Preserving Compression. In Stanley Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 655–663. IEEE Computer Society, 1996.

[8] Andrei Arion, Angela Bonifati, Gianni Costa, Sandra D'Aguanno, Ioana Manolescu, and Andrea Pugliese. XQueC: Pushing Queries to Compressed XML Data. In Freytag et al. [46], pages 1065–1068.

[9] Iliana Avila-Campillo, Todd J. Green, Ashish Gupta, Makoto Onizuka, Demian Raven, and Dan Suciu. XMLTK: An XML Toolkit for Scalable XML Stream Processing. In *Proceedings of PLANX*, 2002.

[10] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the Memory Requirements of XPath Evaluation over XML Streams. In Alin Deutsch, editor, *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 177–188. ACM, 2004.

[11] Denilson Barbosa, Laurent Mignet, and Pierangelo Veltri. Studying the XML Web: Gathering Statistics from an XML Sample. *World Wide Web*, 9(2):187–212, 2006.

[12] Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski. Streaming XPath Processing with Forward and Backward Axes. In Dayal et al. [38], pages 455–466.

[13] Roberto Bayardo Jr., Daniel Gruhl, Vanja Josifovski, and Jussi Myllymaki. An evaluation of binary XML encoding optimizations for fast stream based XML processing. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*, pages 345–354. ACM, 2004.

[14] Stefan Böttcher and Rita Hartel. CSC: Supporting Queries on Compressed Cached XML. In *20th Australasian Database Conference Wellington (ADC-20 2009), 20 - 23 January 2009, New Zealand*, 2009.

[15] Stefan Böttcher, Rita Hartel, and Christian Heinzemann. BSBC: Towards a Succinct Data Format for XML Streams. In José Cordeiro, Joaquim Filipe, and Slimane Hammoudi, editors, *WEBIST 2008, Proceedings of the Fourth International Conference on Web Information Systems and Technologies, Volume 1, Funchal, Madeira, Portugal, May 4-7, 2008*, pages 13–21. INSTICC Press, 2008.

[16] Stefan Böttcher, Rita Hartel, and Christian Messinger. XML stream data reduction by shared KST signatures. In *42nd Hawaii International Conference on Systems Science (HICSS-42 2009), 5-8 January 2009, Waikoloa, Big Island, HI, USA*, 2009.

[17] Stefan Böttcher, Niklas Klein, and Rita Steinmetz. XML Index Compression by DTD Subtraction. In Jorge Cardoso, José Cordeiro, and Joaquim Filipe, editors, *ICEIS 2007 - Proceedings of the Ninth International Conference on Enterprise Information Systems, Volume DISI, Funchal, Madeira, Portugal, June 12-16, 2007*, 2007.

[18] Stefan Böttcher and Rita Steinmetz. A DTD Graph Based XPath Query Subsumption Test. In Zohra Bellahsene, Akmal B. Chaudhri, Erhard Rahm, Michael Rys, and Rainer Unland, editors, *Database and XML Technologies, First International XML Database Symposium, XSym 2003, Berlin, Germany, September 8, 2003, Proceedings*, volume 2824 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2003.

[19] Stefan Böttcher and Rita Steinmetz. Testing Containment of XPath Expressions in Order to Reduce the Data Transfer to Mobile Clients. In Leonid A. Kalinichenko, Rainer Manthey, Bernhard Thalheim, and Uwe Wloka, editors, *Advances in Databases and Information Systems, 7th East European Conference, ADBIS 2003, Dresden, Germany, September 3-6, 2003, Proceedings*, volume 2798 of *Lecture Notes in Computer Science*, pages 400–410. Springer, 2003.

[20] Stefan Böttcher and Rita Steinmetz. DTD-Driven Structure Preserving XML Compression. In David Bell and Jun Hong, editors, *Flexible and Efficient Information Handling, 23rd British National Conference on Databases, BNCOD 23, Belfast, Northern Ireland, UK, July 18-20, 2006, Proceedings*, volume 4042 of *Lecture Notes in Computer Science*, pages 266–269. Springer, 2006.

[21] Stefan Böttcher and Rita Steinmetz. Data Management for Mobile Ajax Web 2.0 Applications. In Roland Wagner, Norman Revell, and Günther Pernul, editors, *Database and Expert Systems Applications, 18th International Conference, DEXA 2007, Regensburg, Germany, September 3-7, 2007, Proceedings*, volume 4653 of *Lecture Notes in Computer Science*, pages 424–433. Springer, 2007.

[22] Stefan Böttcher and Rita Steinmetz. Evaluating XPath Queries on XML Data Streams. In Richard Cooper and Jessie B. Kennedy, editors, *Data Management. Data, Data Everywhere, 24th British National Conference on Databases, BNCOD 24, Glasgow, UK, July 3-5, 2007, Proceedings*, volume 4587 of *Lecture Notes in Computer Science*, pages 101–113. Springer, 2007.

[23] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowa. Extensible Markup Language (XML) 1.1 (Second Edition). http://www.w3.org/TR/2006/REC-xml11-20060816/.

[24] François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, and Markus Spannagel. The XML Stream Query Processor SPEX. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan* [4], pages 1120–1121.

[25] Peter Buneman, Byron Choi, Wenfei Fan, Robert Hutchison, Robert Mann, and Stratis Viglas. Vectorizing and Querying Large XML Repositories. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan* [4], pages 261–272.

[26] Peter Buneman, Martin Grohe, and Christoph Koch. Path Queries on Compressed XML. In Freytag et al. [46], pages 141–152.

[27] Michael Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC, 1994. Research Report 124. 10th May 1994.

[28] Giorgio Busatto, Markus Lohrey, and Sebastian Maneth. Efficient Memory Representation of XML Documents. In Gavin M. Bierman and Christoph Koch, editors, *Database Programming Languages, 10th International Symposium, DBPL 2005, Trondheim, Norway, August 28-29, 2005, Revised Selected Papers*, volume 3774 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2005.

[29] Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors. *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*. Springer, 2002.

[30] K. Selçuk Candan, Wang-Pin Hsiung, Songting Chen, Jun'ichi Tatemura, and Divyakant Agrawal. AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering. In Dayal et al. [39], pages 559–570.

[31] Chee Yong Chan, Pascal Felber, Minos N. Garofalakis, and Rajeev Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA* [1], pages 235–244.

[32] Yi Chen, Susan B. Davidson, and Yifeng Zheng. An Efficient XPath Query Processor for XML Streams. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 79. IEEE Computer Society, 2006.

[33] James Cheney. Compressing XML with Multiplexed Hierarchical PPM Models. In *Data Compression Conference*, pages 163–, 2001.

[34] James Cheng and Wilfred Ng. XQzip: Querying Compressed XML Using Structural Indexing. In Elisa Bertino, Stavros Christodoulakis, Dimitris

Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, volume 2992 of *Lecture Notes in Computer Science*, pages 219–236. Springer, 2004.

[35] Byron Choi. Document Decomposition for XML Compression: A Heuristic Approach. In Mong-Li Lee, Kian-Lee Tan, and Vilas Wuwongse, editors, *DASFAA*, volume 3882 of *Lecture Notes in Computer Science*, pages 202–217. Springer, 2006.

[36] James Clark. RELAX NG Specification. http://relaxng.org/spec-20011203.html.

[37] James Clark and Steve DeRose. XML Path Language (XPath), Version 1.0. http://www.w3.org/TR/xpath.

[38] Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors. *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*. IEEE Computer Society, 2003.

[39] Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors. *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. ACM, 2006.

[40] Alin Deutsch and Val Tannen. Containment and Integrity Constraints for XPath. In Maurizio Lenzerini, Daniele Nardi, Werner Nutt, and Dan Suciu, editors, *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB 2001), Rome, Italy, September 15, 2001*, volume 45 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2001.

[41] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an Internet-Scale XML Dissemination Service. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 612–623. Morgan Kaufmann, 2004.

[42] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. http://www.w3.org/TR/xmlschema-0/.

[43] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In Les Carr,

David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006*, pages 751–760. ACM, 2006.

[44] Damien K. Fisher and Sebastian Maneth. Structural Selectivity Estimation for XML Documents. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, April 15-20, 2007, The Marmara Hotel, Istanbul, Turkey*, pages 626–635. IEEE, 2007.

[45] Massimo Franceschet. XPathMark: Functional and Performance Tests for XPath. In Peter A. Boncz, Torsten Grust, Jérôme Siméon, and Maurice van Keulen, editors, *XQuery Implementation Paradigms*, volume 06472 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[46] Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors. *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*. Morgan Kaufmann, 2003.

[47] Jesse James Garrett. Ajax: A new approach to web applications. http://adaptivepath.com/ideas/essays/archives/000385.php, February 2005. [Online; Stand 18.03.2008].

[48] Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theor. Comput. Sci.*, 368(3):231–246, 2006.

[49] Marc Girardot and Neel Sundaresan. Millau: an encoding format for efficient representation and exchange of XML over the Web. *Computer Networks*, 33(1-6):747–765, 2000.

[50] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China* [2], pages 95–106.

[51] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[52] Ashish Kumar Gupta and Dan Suciu. Stream Processing of XPath Queries with Predicates. In Halevy et al. [53], pages 419–430.

[53] Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM, 2003.

[54] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) Level 3 Core Specification, Version 1.0. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/.

[55] D. A. Huffman. A method for construction of minimum-redundancy codes. *Proceedings IRE*, 40:1098–1101, 1952.

[56] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. An XML query engine for network-bound data. *VLDB J.*, 11(4):380–402, 2002.

[57] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *VLDB J.*, 14(2):197–210, 2005.

[58] Laks V. S. Lakshmanan, Hui Wang, and Zheng (Jessica) Zhao. Answering Tree Pattern Queries Using Views. In Dayal et al. [39], pages 571–582.

[59] Christopher League and Kenjone Eng. Type-Based Compression of XML Data. In *2007 Data Compression Conference (DCC 2007), 27-29 March 2007, Snowbird, UT, USA*, pages 273–282. IEEE Computer Society, 2007.

[60] Hartmut Liefke and Dan Suciu. XMILL: An Efficient Compressor for XML Data. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 153–164. ACM, 2000.

[61] Bhushan Mandhani and Dan Suciu. Query Caching and View Selection for XML Databases. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 469–480. ACM, 2005.

[62] Gerome Miklau and Dan Suciu. Containment and Equivalence for an XPath Fragment. In Lucian Popa, editor, *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 3-5, Madison, Wisconsin, USA*, pages 65–76. ACM, 2002.

[63] Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. XPRESS: A Queriable Compression for XML Data. In Halevy et al. [53], pages 122–133.

[64] Frank Neven and Thomas Schwentick. XPath Containment in the Presence of Disjunction, DTDs, and Variables. In Calvanese et al. [29], pages 312–326.

[65] Craig G. Nevill-Manning and Ian H. Witten. Identifying Hierarchical Strcture in Sequences: A Linear-Time Algorithm. *J. Artif. Intell. Res. (JAIR)*, 7:67–82, 1997.

[66] Wilfred Ng, Wai Yeung Lam, Peter T. Wood, and Mark Levene. XCQ: A queriable XML compression system. *Knowl. Inf. Syst.*, 10(4):421–452, 2006.

[67] Dan Olteanu, Tobias Kiesling, and François Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. In Dayal et al. [38], pages 702–704.

[68] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In Akmal B. Chaudhri, Rainer Unland, Chabane Djeraba, and Wolfgang Lindner, editors, *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops, EDBT 2002 Workshops XMLDM, MDDE, and YRWS, Prague, Czech Republic, March 24-28, 2002, Revised Papers*, volume 2490 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 2002.

[69] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 903–908. ACM, 2004.

[70] Linda Dailey Paulson. Building Rich Web Applications with Ajax. *Computer*, 38(10):14–17, 2005.

[71] Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data. In Halevy et al. [53], pages 431–442.

[72] Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China* [2], pages 974–985.

[73] Hariharan Subramanian and Priti Shankar. Compressing XML Documents Using Recursive Finite State Automata. In Jacques Farré, Igor

Litovsky, and Sylvain Schmitz, editors, *Implementation and Application of Automata, 10th International Conference, CIAA 2005, Sophia Antipolis, France, June 27-29, 2005, Revised Selected Papers*, volume 3845 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2005.

[74] Neel Sundaresan and Reshad Moussa. Algorithms and programming models for efficient representation of XML for Internet applications. In *Proceedings of the Tenth International World Wide Web Conference, WWW 10, Hong Kong, China*, pages 366–375. ACM, 2001.

[75] Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A Query-Friendly XML Compressor. In *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA* [1], pages 225–234.

[76] Christian Werner, Carsten Buschmann, Ylva Brandt, and Stefan Fischer. Compressing SOAP Messages by using Pushdown Automata. In *2006 IEEE International Conference on Web Services (ICWS 2006), 18-22 September 2006, Chicago, Illinois, USA*, pages 19–28. IEEE Computer Society, 2006.

[77] Peter T. Wood. Containment for XPath Fragments under DTD Constraints. In Calvanese et al. [29], pages 297–311.

[78] Benjamin Bin Yao, M. Tamer Özsu, and Nitin Khandelwal. XBench Benchmark and Performance Testing of XML DBMSs. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA* [3], pages 621–633.

[79] Ning Zhang, Varun Kacholia, and M. Tamer Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA* [3], pages 54–65.

[80] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.